Information Systems 34 (2009) 209–230

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Multi-query optimization for sketch-based estimation ☆

Alin Dobra [a,*], Minos Garofalakis [b], Johannes Gehrke [c], Rajeev Rastogi [d]

[a] University of Florida, Gainesville, FL 32611, USA
[b] Yahoo! Research, 2821 Mission College Blvd. Santa Clara, CA 95054, USA
[c] 4105B Upson Hall Cornell University Ithaca, NY 14853, USA
[d] Bell Laboratories Rm 2B-301, 700 Mountain Avenue Murray Hill, NJ 07974, USA

ARTICLE INFO

Article history:
Received 3 July 2007
Received in revised form
15 May 2008
Accepted 20 June 2008
Recommended by K.A. Ross

Keywords:
Data streaming
Sketches
Approximate query processing
Multi-query optimization

ABSTRACT

Randomized techniques, based on computing small "sketch" synopses for each stream, have recently been shown to be a very effective tool for approximating the result of a single SQL query over streaming data tuples. In this paper, we investigate the problems arising when data-stream sketches are used to process *multiple* such queries concurrently. We demonstrate that, in the presence of multiple query expressions, intelligently sharing sketches among concurrent query evaluations can result in substantial improvements in the utilization of the available sketching space and the quality of the resulting approximation error guarantees. We provide necessary and sufficient conditions for multi-query sketch sharing that guarantee the correctness of the result-estimation process. We also investigate the difficult optimization problem of determining sketch-sharing configurations that are optimal (e.g., under a certain error metric for a given amount of space). We prove that optimal sketch sharing typically gives rise to $\mathcal{NP}$-hard questions, and we propose novel heuristic algorithms for finding good sketch-sharing configurations in practice. Results from our experimental study with queries from the TPC-H benchmark verify the effectiveness of our approach, clearly demonstrating the benefits of our sketch-sharing methodology.

1. Introduction

Traditional Database Management Systems (DBMS) software is built on the concept of *persistent* data sets that are stored reliably in stable storage and queried several times throughout their lifetime. For several emerging application domains, however, data arrive and need to be processed continuously, without the benefit of several passes over a static, persistent data image. Such *continuous data streams* arise naturally, for example, in the network installations of large telecom and Internet service providers where detailed usage information (call-detail-records, SNMP/RMON packet-flow data, etc.) from different parts of the underlying network needs to be continuously collected and analyzed for interesting trends. Other applications that generate rapid-rate and massive volumes of stream data include retail-chain transaction processing, ATM and credit card operations, financial tickers, Web-server activity logging, and so on. In most such applications, the data stream is actually accumulated and archived in the DBMS of a (perhaps, off-site) data warehouse, often making access to the archived data prohibitively expensive. Further, the ability to make decisions and infer interesting patterns *on-line* (i.e., as the data stream arrives) is crucial for several mission-critical tasks that can have significant dollar value for a large corporation (e.g., telecom fraud detection). As a result, there has been increasing interest in

☆ This is an extended version of the paper *Sketch-based multi-query processing over data streams* that appeared in Proceedings of EDBT 2004 conference.

* Corresponding author. Tel.: +1 352 514 0743.
E-mail addresses: adobra@cise.ufl.edu (A. Dobra), minos@yahoo-inc.com (M. Garofalakis), johannes@cs.cornell.edu (J. Gehrke), rastogi@bell-labs.com (R. Rastogi).

0306-4379/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2008.06.002

designing data-processing algorithms that work over continuous data streams, i.e., algorithms that provide results to user queries while looking at the relevant data items *only once and in a fixed order* (determined by the stream-arrival pattern).

Given the large diversity of users and/or applications that a generic query-processing environment typically needs to support, it is evident that any realistic stream-query processor must be capable of effectively handling *multiple* standing queries over a collection of input data streams. Given a collection of queries to be processed over incoming streams, two key effectiveness parameters are (1) the amount of *memory* made available to the on-line algorithm and (2) the *per-item processing time* required by the query processor.

Memory, in particular, constitutes an important constraint on the design of stream processing algorithms since, in a typical streaming environment, only limited memory resources are made available to each of the standing queries. In these situations, we need algorithms that can summarize the data streams involved in concise *synopses* that can be used to provide *approximate answers* to user queries along with some reasonable guarantees on the quality of the approximation. Such approximate, on-line query answers are particularly well-suited to the exploratory nature of most data-stream processing applications such as, e.g., trend analysis and fraud/anomaly detection in telecom-network data, where the goal is to identify generic, interesting or "out-of-the-ordinary" patterns rather than provide results that are exact to the last decimal.

Example 1.1. Consider a fault-management application running in a large IP-network installation that provides client applications (e.g., Web browsers) on host machines with access to application servers (e.g., Web servers, SAP, Oracle) through paths of IP routers and switches. During operation, network entities continuously generate streams of *alarms* to signal a variety of abnormal conditions. For example, a client application that cannot connect to a server typically generates an alarm which contains (in addition to other attributes), a timestamp and an alarm_type. The timestamp is essentially the time at which the alarm is generated by the end application and is of a sufficiently coarse granularity (e.g., seconds or minutes) to enable meaningful cross-correlation with other network events. The alarm_type indicates the reason for the alarm which could range from the client connection to the server timing out, to simply a daemon process dying on the host machine. Similarly, each network element (router or switch) can also generate a variety of alarms with similar attributes in the event of a link outage to a neighboring element, excessive congestion, disproportionate number of packet errors at an interface, and so on.

These alarm streams are routed to the fault-analysis platform running at a central *Network Operations Center (NOC)* and are typically archived in an alarm warehouse for off-line analysis. One of the primary goals of fault-analysis software is to be able to quickly pinpoint the *root cause* of an observed fault in the network. Fast root-cause inference, in turn, depends crucially on the ability to effectively correlate the observed streams of alarm signals from various network entities based on their attribute values. For instance, suppose we are interested in knowing if the frequent occurrence of similar (i.e., same alarm_type) problems at routers R_1 and R_2 (e.g., link outage) is responsible for the inability of client application C to connect to a server. To answer this question, we would like to estimate the number of times C, R_1, and R_2 produce alarm signals with identical timestamps and alarm types; a large number of such events would allow us to safely establish this correlation. Similarly, we are also interested in knowing whether problems observed at router R_1 are correlated with similar problems observed at a third router R_3; this, again, gives rise to a similar count-estimation query. More concretely, if C, R_1, R_2, and R_3 denote the alarm data streams for the four network entities, then we would like to estimate the result of the following two SQL join queries Q_1, Q_2 over the four streams: $Q_1 =$ "SELECT COUNT FROM C, R_1, R_2 WHERE C.timestamp $= R_1$.timestamp AND C.timestamp $= R_2$.timestamp AND R_1.alarm_type $= R_2$.alarm_type" and $Q_2 =$ "SELECT COUNT FROM R_2, R_3 WHERE R_2.timestamp $= R_3$.timestamp AND R_2.alarm_type $= R_3$.alarm_type". Of course, given the stringent resource constraints and high volume of event arrivals at the NOC, processing such stream queries is feasible only if it is guaranteed to consume limited memory and CPU cycles.

1.1. Prior work

The recent surge of interest in data-stream computation has led to several (theoretical and practical) studies proposing novel one-pass algorithms with limited memory requirements for different problems; examples include: quantile and order-statistics computation [1,2]; distinct-element counting [3–5]; frequent itemset counting [6,7]; estimating frequency moments, join sizes, and difference norms [8–11]; data clustering and decision-tree construction [12,13]; estimating correlated aggregates [14]; and computing one- or multi-dimensional histograms or Haar wavelet decompositions [15,16]. All these papers rely on an approximate query-processing model, typically based on an appropriate underlying synopsis data structure. (A different approach, explored by the Stanford STREAM project [17], is to characterize a subclass of queries that can be computed *exactly* with bounded memory.) The synopses of choice for a number of the above-cited papers are based on the key idea of *pseudo-random sketches* which, essentially, can be thought of as simple, randomized linear projections of the underlying data vector(s) [18]. Dobra et al. [19] demonstrated the utility of sketch synopses in computing provably-accurate approximate answers for a *single* SQL query comprising (possibly) multiple join operators—the current work is the extension to multiple SQL queries.

None of these earlier research efforts has addressed the more general problem of effectively providing accurate

approximate answers to *multiple* SQL queries over a collection of input streams. Of course, the problem of *multi-query optimization* (that is, optimizing multiple queries for concurrent execution in a conventional DBMS) has been around for some time, and several techniques for extending conventional query optimizers to deal with multiple queries have been proposed [20,21]. The cornerstone of all these techniques is the discovery of common query sub-expressions whose evaluation can be shared among the query-execution plans produced. Very similar ideas have also found their way in large-scale, continuous-query systems (e.g., NiagaraCQ [22]) that try to optimize the evaluation of large numbers of trigger conditions. As will become clear later, however, approximate multi-query processing over streams with limited space gives rise to several novel and difficult optimization issues that are very different from those of traditional multi-query optimization.

1.2. Our contributions

In this paper, we tackle the problem of efficiently processing multiple (possibly, multi-join) concurrent aggregate SQL queries over a collection of input data streams. Similar to earlier work on data streaming [8,19], our approach is based on computing small, pseudo-random sketch synopses of the data. We demonstrate that, in the presence of multiple query expressions, intelligently *sharing sketches* among concurrent (approximate) query evaluations can result in substantial improvements in the utilization of the available sketching space and the quality of the resulting approximation error guarantees. We provide necessary and sufficient conditions for multi-query sketch sharing that guarantee the correctness of the resulting sketch-based estimators. We also attack the difficult optimization problem of determining sketch-sharing configurations that are optimal (e.g., under a certain error metric for a given amount of space). We prove that optimal sketch sharing typically gives rise to $\mathcal{NP}$-hard questions, and we propose novel heuristic algorithms for finding effective sketch-sharing configurations in practice. More concretely, the key contributions of our work can be summarized as follows.

- *Multi-query sketch sharing*: *concepts and conditions.* We formally introduce the concept of *sketch sharing* for efficient, approximate multi-query stream processing. Briefly, the basic idea is to share sketch computation and sketching space across several queries in the workload that can effectively use the same sketches over (a subset of) their input streams. Of course, since sketches and sketch-based estimators are probabilistic in nature, we also need to ensure that this sharing does not degrade the correctness and accuracy of our estimates by causing desirable estimator properties (e.g., unbiasedness) to be lost. Thus, we present necessary and sufficient conditions (based on the resulting multi-join graph) that fully characterize such "correct" sketch-sharing configurations for a given query workload.
- *Novel sketch-sharing optimization problems and algorithms.* Given that multiple correct sketch-sharing configurations can exist for a given stream-query workload, our processor should be able to identify configurations that are optimal or near-optimal; for example, under a certain (aggregate) error metric for the workload and for a given amount of sketching space. We formulate these sketch-sharing optimization problems for different metrics of interest, and propose novel algorithmic solutions for the two key sub-problems involved, namely: (1) *space allocation*: determine the best amount of space to be given to each sketch for a fixed sketch-sharing configuration under two different metrics, average error and maximum error; and (2) *join coalescing*: determine an optimal sketch-sharing plan by deciding which joins in the workload will share sketches. For the space allocation problem with the average error metric, we show that the allocation problem is $\mathcal{NP}$-hard but a provable good approximation can be obtained by rounding the solution of the continuous problem. To solve the continuous problem we employ optimization theory tools to provide an efficient custom solution based on reductions to Network Flow problems. The space allocation problem under the maximum metric turns out to be much simpler—it can be solved exactly essentially in linear time. Solutions to these two problems are used to provide heuristic approximations to the join coalescing problem, problem that is $\mathcal{NP}$-hard.
- *Implementation results validating our sketch-sharing techniques.* We present the results from an empirical study of our sketch-sharing schemes with several synthetic data sets and multi-query workloads based on the TPC-H benchmark. Our results clearly demonstrate the benefits of effective sketch-sharing over realistic query workloads, showing that significant improvements in answer quality are possible compared to a naive, no-sharing approach. Specifically, our experiments indicate that sketch sharing can boost accuracy of query answers by factors ranging from 2–4 for a wide range of multi-query workloads.

2. Streams and random sketches

2.1. Stream data-processing model

We now briefly describe the key elements of our generic architecture for multi-query processing over continuous data streams (depicted in Fig. 1); similar architectures (for the single-query setting) have been described elsewhere (e.g., [19,15]). Consider a workload $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ comprising a collection of arbitrary (complex) SQL queries $Q_1, \ldots, Q_q$ over a set of relations $R_1, \ldots, R_r$ (of course, each query typically references a subset of the relations/attributes in the input). Also, let $|R_i|$ denote the total number of tuples in R_i. In contrast to conventional DBMS query processors, our stream query-processing engine is allowed to see the data tuples in $R_1, \ldots, R_r$ *only once* and in fixed order as they are

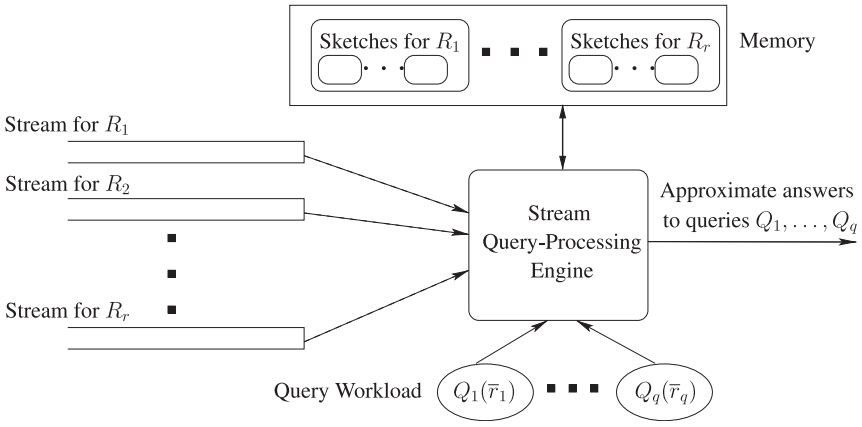

Fig. 1. Stream multi-query processing architecture.

streaming in from their respective source(s). Backtracking over the data stream and explicit access to past data tuples are impossible. Further, the order of tuple arrival for each relation R_i is arbitrary and duplicate tuples can occur anywhere over the duration of the R_i stream. (Our techniques can also readily handle tuple *deletions* in the streams.) Hence, our stream data model assumes the most general "unordered, cash-register" rendition of stream data considered by Gilbert et al. [15] for computing one-dimensional Haar wavelets over streaming values and, of course, generalizes their model to multiple, multi-dimensional streams since each R_i can comprise several distinct attributes.

Our stream query-processing engine is also allowed a certain amount of memory, typically significantly smaller than the total size of the data set(s). This memory is used to maintain a set of concise *synopses* for each data stream R_i. The key constraints imposed on such synopses are that: (1) they are much smaller than the total number of tuples in R_i (e.g., their size is logarithmic or polylogarithmic in $|R_i|$) and (2) they can be computed quickly, in a single pass over the data tuples in R_i in the (arbitrary) order of their arrival. At any point in time, our query-processing algorithms can combine the maintained collection of synopses to produce approximate answers to all queries in $\mathcal{Q}$.

For ease of reference, Table 1 summarizes some of the key notational conventions used in this paper. Detailed definitions are provided at the appropriate locations in the text.

Table 1
Notation

Symbol	Semantics
R_i	Input data stream ($i = 1, \ldots, r$)
$R_i.A_j$	jth attribute of relation R_i
$\mathcal{Q}$	Query workload comprising queries $\{Q_1, \ldots, Q_q\}$
Q	Generic query in $\mathcal{Q}$
$\mathcal{E}$	Generic equi-join condition
n	Number of equi-join constraints in $\mathcal{E}$
$\mathcal{J}(Q)$ ($\mathcal{J}(\mathcal{Q})$)	Join graph(s) for query Q (set of queries $\mathcal{Q}$)
$v_1, \ldots, v_k$	Join-graph vertices
$\mathcal{A}(v)$	Attributes of vertex v that appear in $\mathcal{E}$
X_v	atomic sketch for vertex v
$Q(v)$	Set of queries attached to vertex v
$\mathcal{J}(\mathcal{Q})$	family of (possible) join graphs for queries $\mathcal{Q}$
$V(Q_i)$	Subset of vertices used in query Q_i
M_{Q_i}	Number of iid copies of estimator X_{Q_i} for Q_i
W_{Q_i}	$W_{Q_i} = \frac{8\mathrm{Var}[X_{Q_i}]}{E[X_{Q_i}]^2}$, a constant for each query Q_i

2.2. Approximating single-query answers with pseudo-random sketch summaries

The basic technique: *binary-join size tracking* [8,9]. Consider a simple stream-processing scenario where the goal is to estimate the size of a binary join of two streams R_1 and R_2 on attributes $R_1.A_1$ and $R_2.A_2$, respectively. That is, we seek to approximate the result of query $Q = \mathtt{COUNT}(R_1 \bowtie_{R_1.A_1=R_2.A_2} R_2)$ as the tuples of R_1 and R_2 are streaming in. Let $\mathtt{dom}(A)$ denote the domain of an attribute A [1] and $f_R(i)$ be the frequency of attribute value i in $R.A$. (Note that, by the definition of the equi-join operator, the two join attributes have identical value domains, i.e., $\mathtt{dom}(A_1) = \mathtt{dom}(A_2)$). Thus, we want to produce an estimate for the expression $Q = \sum_{i \in \mathtt{dom}(A_1)} f_{R_1}(i) f_{R_2}(i)$. Clearly, estimating this join size exactly requires at least $\Omega(|\mathtt{dom}(A_1)|)$ space, making an exact solution impractical for a data-stream setting. In their seminal work, Alon et al. [8,9] propose a randomized technique that can offer strong probabilistic guarantees on the quality of the resulting join-size estimate while using only logarithmic space in $|\mathtt{dom}(A_1)|$.

Briefly, the basic idea of their scheme is to define a random variable X_Q that can be easily computed over the streaming values of $R_1.A_1$ and $R_2.A_2$, such that (1) X_Q is an *unbiased* (i.e., correct on expectation) estimator for the

[1] Without loss of generality, we assume that each attribute domain $\mathtt{dom}(A)$ is indexed by the set of integers $\{1, \ldots, |\mathtt{dom}(A)|\}$, where $|\mathtt{dom}(A)|$ denotes the size of the domain.

target join size, so that $E[X_Q] = Q$; and (2) X_Q's variance ($\mathrm{Var}(X_Q)$) can be appropriately upper-bounded to allow for probabilistic guarantees on the quality of the Q estimate. This random variable X_Q is constructed on-line from the two data streams as follows:

- Select a family of *four-wise independent binary random variables* $\{\xi_i : i = 1, \ldots, |\mathrm{dom}(A_1)|\}$, where each $\xi_i \in \{-1, +1\}$ and $P[\xi_i = +1] = P[\xi_i = -1] = \frac{1}{2}$ (i.e., $E[\xi_i] = 0$). Informally, the four-wise independence condition means that for any 4-tuple of ξ_i variables and for any 4-tuple of $\{-1, +1\}$ values, the probability that the values of the variables coincide with those in the $\{-1, +1\}$ 4-tuple is exactly $\frac{1}{16}$ (the product of the equality probabilities for each individual ξ_i). The crucial point here is that, by employing known tools (e.g., orthogonal arrays) for the explicit construction of small sample spaces supporting four-wise independent random variables, such families can be efficiently constructed on-line using only $O(\log |\mathrm{dom}(A_1)|)$ space [9].
- Define $X_Q = X_1 \cdot X_2$, where $X_k = \sum_{i \in \mathrm{dom}(A_1)} f_{R_k}(i)\xi_i$, for $k = 1, 2$. Note that each X_k is simply a randomized linear projection (inner product) of the frequency vector of $R_k.A_k$ with the vector of ξ_i's that can be efficiently generated from the streaming values of A_k as follows: start with $X_k = 0$ and simply add ξ_i to X_k whenever the ith value of A_k is observed in the stream.

The quality of the estimation guarantees can be improved using a standard *boosting technique* that maintains several independent identically distributed (iid) instantiations of the above process, and uses averaging and median-selection operators over the X_Q estimates to boost accuracy and probabilistic confidence [9]. (Independent instances can be constructed by simply selecting independent random seeds for generating the families of four-wise independent ξ_i's for each instance.) We use the term *atomic sketch* to describe each randomized linear projection computed over a data stream. Letting $\mathrm{SJ}_k (k = 1, 2)$ denote the self-join size of $R_k.A_k$ (i.e., $\mathrm{SJ}_k = \sum_{i \in \mathrm{dom}(A_k)} f_{R_k}(i)^2$), the following theorem [8] shows how sketching can be applied for estimating binary-join sizes in limited space. (By standard Chernoff bounds, using median-selection over $O(\log(1/\delta))$ of the averages computed in Theorem 1 allows the confidence in the estimate to be boosted to $1 - \delta$, for any pre-specified $\delta < 1$.)

Theorem 1 (*Alon et al. [8]*). *Let the atomic sketches X_1 and X_2 be as defined above. Then $E[X_Q] = E[X_1 X_2] = Q$ and $\mathrm{Var}(X_Q) \leqslant 2 \cdot \mathrm{SJ}_1 \cdot \mathrm{SJ}_2$. Thus, averaging the X_Q estimates over $O((\mathrm{SJ}_1 \mathrm{SJ}_2 / Q^2 \varepsilon^2) \log \frac{1}{\delta})$ i.i.d. instantiations of the basic scheme, guarantees an estimate that lies within a relative error of ε from Q with probability at least $1 - \delta$.*

Single multi-join query answering [19]. In more recent work, Dobra et al. [19] have extended sketch-based techniques to approximate the result of a general, multi-join aggregate SQL query over a collection of streams.[2] More specifically, they focus on approximating a multi-join stream query Q of the form: "SELECT COUNT FROM $R_1, R_2, \ldots, R_r$ WHERE $\mathscr{E}$", where $\mathscr{E}$ represents the conjunction of n equi-join constraints of the form $R_i.A_j = R_k.A_l$ ($R_i.A_j$ denotes the jth attribute of relation R_i). (The extension to other aggregate functions, e.g., SUM, is fairly straightforward [19].) Their development also assumes that each attribute $R_i.A_j$ appears in $\mathscr{E}$ at most once; this requirement can be easily achieved by simply renaming repeating attributes in the query. In what follows, we describe the key ideas and results from [19] based on the join-graph model of the input query Q, since this will allow for a smoother transition to the multi-query case (Section 3).

We assume that all R_i are distinct, and that each $R_i.A_j$ appears in $\mathscr{E}$ at most once. Note that this can be easily achieved by renaming duplicate relations and relation, attribute pairs in the query; thus, two different relation names in the query may correspond to the same underlying relation stream, and the same attribute in a relation may occur in $\mathscr{E}$ with different names. For instance, the query SELECT COUNT FROM R_1, R_1 WHERE $R_1.A_1 = R_1.A_1$, after relation renaming becomes SELECT COUNT FROM R_1, R_2 WHERE $R_1.A_1 = R_2.A_1$ (here R_1 and R_2 refer to the same underlying relation stream). Similarly, attribute renaming causes the query SELECT COUNT FROM R_1, R_2, R_3 WHERE $R_1.A_1 = R_2.A_1 \wedge R_2.A_1 = R_3.A_1$ to be transformed to the following new rewritten query: SELECT COUNT FROM R_1, R_2, R_3 WHERE $R_1.A_1 = R_2.A_1 \wedge R_2.A_2 = R_3.A_1$ (here $R_2.A_1$ and $R_2.A_2$ refer to the same underlying attribute of relation R_2).

Given stream query Q, we define the *join graph* of Q (denoted by $\mathscr{J}(Q)$), as follows. There is a distinct vertex v in $\mathscr{J}(Q)$ for each stream R_i referenced in Q (we use $R(v)$ to denote the relation associated with vertex v). For each equality constraint $R_i.A_j = R_k.A_l \in \mathscr{E}$, we add a distinct undirected edge $e = \langle v, w \rangle$ to $\mathscr{J}(Q)$, where $R(v) = R_i$ and $R(w) = R_k$; we also label this edge with the triple $\langle R_i.A_j, R_k.A_l, Q \rangle$ that specifies the attributes in the corresponding equality constraint and the enclosing query Q (the query label is used in the multi-query setting). Given an edge $e = \langle v, w \rangle$ with label $\langle R_i.A_j, R_k.A_l, Q \rangle$, the three components of e's label triple can be obtained as $A_v(e), A_w(e)$, and $Q(e)$. (Clearly, by the definition of equi-joins, $\mathrm{dom}(A_v(e)) = \mathrm{dom}(A_w(e))$.) Note that there may be multiple edges between a pair of vertices in the join graph, but each edge has its own distinct label triple. Finally, for a vertex v in $\mathscr{J}(Q)$, we denote the attributes of $R(v)$ that appear in the input query (or, queries) as $\mathscr{A}(v)$; thus, $\mathscr{A}(v) = \{A_v(e) : \text{edge } e \text{ is incident on } v\}$.

The result of Q is the number of tuples in the cross-product of $R_1, \ldots, R_r$ that satisfy the equality constraints in $\mathscr{E}$ over the join attributes. Similar to the basic sketching method [8,9], the algorithm of Dobra et al. constructs an unbiased, bounded-variance probabilistic estimate X_Q for Q using atomic sketches built on the vertices of the join

[2] [19] also describes a *sketch-partitioning* technique for improving the quality of basic sketching estimates; this technique is essentially orthogonal to the multi-query sketch-sharing problem considered in this paper, so we do not discuss it further.

graph $\mathcal{J}(Q)$. More specifically, for each edge $e = \langle v, w\rangle$ in $\mathcal{J}(Q)$, their algorithm defines a family of four-wise independent random variables $\xi^e = \{\xi_i^e : i = 1, \ldots, |\mathrm{dom}(A_v(e))|\}$, where each $\xi_i^e \in \{-1, +1\}$. The key here is that the equi-join attribute pair $A_v(e), A_w(e)$ associated with edge e shares the same ξ family; on the other hand, distinct edges of $\mathcal{J}(Q)$ use *independently generated* ξ families (using mutually independent random seeds). The atomic sketch X_v for each vertex v in $\mathcal{J}(Q)$ is built as follows. Let $e_1, \ldots, e_k$ be the edges incident on v and, for $i_1 \in \mathrm{dom}(A_v(e_1)), \ldots, i_k \in \mathrm{dom}(A_v(e_k))$, let $f_v(i_1, \ldots, i_k)$ denote the number of tuples in $R(v)$ that match values $i_1, \ldots, i_k$ in their join attributes. More formally, $f_v(i_1, \ldots, i_k)$ is the number of tuples $t \in R(v)$ such that $t[A_v(e_j)] = i_j$, for $1 \leqslant j \leqslant k$ ($t[A]$ denotes the value of attribute A in tuple t). Then, the atomic sketch at v is $X_v = \sum_{i_1 \in \mathrm{dom}(A_v(e_1))} \cdots \sum_{i_k \in \mathrm{dom}(A_v(e_k))} f_v(i_1, \ldots, i_k) \prod_{j=1}^{k} \xi_{i_j}^{e_j}$. Finally, the estimate for Q is defined as $X_Q = \prod_v X_v$ (that is, the product of the atomic sketches for all vertices in $\mathcal{J}(Q)$). Note that each atomic sketch X_v can be efficiently computed as tuples of $R(v)$ are streaming in; more specifically, X_v is initialized to 0 and, for each tuple t in the $R(v)$ stream, the quantity $\prod_{j=1}^{k} \xi_{t[A_v(e_j)]}^{e_j}$ is added to X_v.

Example 2.1. Consider query $Q =$ SELECT COUNT FROM R_1, R_2, R_3 WHERE $R_1.A_1 = R_2.A_1$ AND $R_2.A_2 = R_3.A_2$. The join graph $\mathcal{J}(Q)$ is depicted in Fig. 2, with vertices v_1, v_2, and v_3 corresponding to streams R_1, R_2, and R_3, respectively. Similarly, edges e_1 and e_2 correspond to the equi-join constraints $R_1.A_1 = R_2.A_1$ and $R_2.A_2 = R_3.A_2$, respectively. (Just to illustrate our notation, $R(v_1) = R_1, A_{v_2}(e_1) = R_2.A_1$ and $\mathcal{A}(v_2) = \{R_2.A_1, R_2.A_2\}$.) The sketch construction defines two families of four-wise independent random families (one for each edge): $\{\xi_i^{e_1}\}$ and $\{\xi_j^{e_2}\}$. The three atomic sketches X_{v_1}, X_{v_2}, and X_{v_3} (one for each vertex) are defined as $X_{v_1} = \sum_{i \in \mathrm{dom}(R_1.A_1)} f_{v_1}(i)\ \xi_i^{e_1}$, $X_{v_2} = \sum_{i \in \mathrm{dom}(R_2.A_1)} \sum_{j \in \mathrm{dom}(R_2.A_2)} f_{v_2}(i,j)\ \xi_i^{e_1} \xi_j^{e_2}$, and $X_{v_3} = \sum_{j \in \mathrm{dom}(R_3.A_2)} f_{v_3}(j)\ \xi_j^{e_3}$. The value of the random variable $X_Q = X_{v_1} X_{v_2} X_{v_3}$ gives the sketching estimate for the result of Q.

Dobra et al. [19] demonstrate that the random variable X_Q constructed above is an unbiased estimator for Q, and prove the following theorem which generalizes the earlier result of Alon et al. to multi-join queries. ($\mathrm{SJ}_v = \sum_{i_1 \in \mathrm{dom}(A_v(e_1))} \cdots \sum_{i_k \in \mathrm{dom}(A_v(e_k))} f_v(i_1, \ldots, i_k)^2$ is the self-join size of $R(v)$.)

Theorem 2.2. *Let Q be a* COUNT *query with n equi-join predicates such that $\mathcal{J}(\mathcal{Q})$ contains no cycles of length >2. Then, $E[X_Q] = Q$ and using sketching space of $O(\mathrm{Var}[X_Q] \cdot \log(1/\delta)/Q^2 \cdot \varepsilon^2)$, it is possible to approximate Q to within a relative error of ε with probability at least $1 - \delta$, where $\mathrm{Var}[X_Q] \leqslant 2^{2n} \prod_v \mathrm{SJ}_v$.*

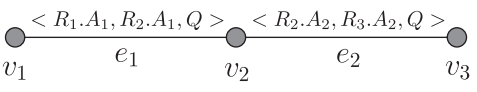

Fig. 2. Example query join graph.

3. Sketch sharing: basic concepts and problem formulation

In this section, we turn our attention to sketch-based processing of *multiple* aggregate SQL queries over streams. We introduce the basic idea of sketch sharing and demonstrate how it can improve the effectiveness of the available sketching space and the quality of the resulting approximate answers. We also characterize the class of correct sketch-sharing configurations and formulate the optimization problem of identifying an effective sketch-sharing plan for a given query workload.

3.1. Sketch sharing

Consider the problem of using sketch synopses for the effective processing of a query workload $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ comprising multiple (multi-join) COUNT aggregate queries. As in [19], we focus on COUNT since the extension to other aggregate functions is relatively straightforward; we also assume an attribute-renaming step that ensures that each stream attribute is referenced only once in each of the Q_is (of course, the same attribute can be used multiple times across the queries in $\mathcal{Q}$).

An obvious solution to our multi-query processing problem is to build disjoint join graphs $\mathcal{J}(Q_i)$ for each query $Q_i \in \mathcal{Q}$, and construct independent atomic sketches for the vertices of each $\mathcal{J}(Q_i)$. The atomic sketches for each vertex of $\mathcal{J}(Q_i)$ can then be combined to compute an approximate answer for Q_i as described in [19] (Section 2.2). A key drawback of such a naive solution is that it ignores the fact that a relation R_i may appear in multiple queries in $\mathcal{Q}$. Thus, it should be possible to reduce the overall space requirements by *sharing* atomic-sketch computations among the vertices for stream R_i in the join graphs for the queries in our workload. We illustrate this in the following example.

Example 3.1. Consider queries $Q_1 =$ SELECT COUNT FROM R_1, R_2, R_3 WHERE $R_1.A_1 = R_2.A_1$ AND $R_2.A_2 = R_3.A_2$ and $Q_2 =$ SELECT COUNT FROM R_1, R_3 WHERE $R_1.A_1 = R_3.A_2$. The naive processing algorithm described above would maintain two disjoint join graphs (Fig. 3) and, to compute a single pair (X_{Q_1}, X_{Q_2}) of sketch-based estimates, it would use three families of random variables (ξ^{e_1}, ξ^{e_2}, and ξ^{e_3}), and a total of five atomic sketches ($X_{v_k}, k = 1, \ldots, 5$).

Instead, suppose that we decide to re-use the atomic sketch X_{v_1} for v_1 also for v_4, both of which essentially correspond to the same attribute of the same stream ($R_1.A_1$). Since for each $i \in \mathrm{dom}(R_1.A_1)$, $f_{v_4}(i) = f_{v_1}(i)$, we get $X_{v_4} = X_{v_1} = \sum_{i \in \mathrm{dom}(R_1.A_1)} f_{v_4}(i) \xi_i^{e_1}$. Of course, in order to

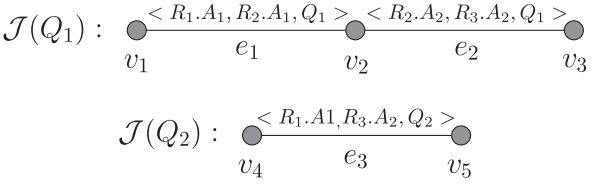

Fig. 3. Example workload with sketch-sharing potential.

correctly compute a probabilistic estimate of Q_2, we also need to use the same family ξ^{e_1} in the computation of X_{v_5}; that is, $X_{v_5} = \sum_{i \in \mathrm{dom}(R_1.A_1)} f_{v_5}(i)\xi_i^{e_1}$. It is easy to see that both final estimates $X_{Q_1} = X_{v_1}X_{v_2}X_{v_3}$ and $X_{Q_1} = X_{v_1}X_{v_5}$ satisfy all the premises of the sketch-based estimation results in [19]. Thus, by simply sharing the atomic sketches for v_1 and v_4, we have reduced the total number of random families used in our multi-query processing algorithm to two (ξ^{e_1} and ξ^{e_2}) and the total number of atomic sketches maintained to four.

Let $\mathcal{J}(\mathcal{Q})$ denote the collection of all join graphs in workload $\mathcal{Q}$, i.e., all $\mathcal{J}(Q_i)$ for $Q_i \in \mathcal{Q}$. Sharing sketches between the vertices of $\mathcal{J}(\mathcal{Q})$ can be seen as a transformation of $\mathcal{J}(\mathcal{Q})$ that essentially *coalesces* in vertices belonging to different join graphs in $\mathcal{J}(\mathcal{Q})$. (We also use $\mathcal{J}(\mathcal{Q})$ to denote the transformed multi-query join graph.) Of course, as shown in Example 3.1, vertices $v \in \mathcal{J}(Q_i)$ and $w \in \mathcal{J}(Q_j)$ can be coalesced in this manner *only if* $R(v) = R(w)$ (i.e., they correspond to the same data stream) and $\mathcal{A}(v) = \mathcal{A}(w)$ (i.e., both Q_i and Q_j use exactly the same attributes of that stream). Such vertex coalescing implies that a vertex v in $\mathcal{J}(\mathcal{Q})$ can have edges from multiple different queries incident on it; we denote the set of all these queries as $Q(v)$, i.e., $Q(v) = \{Q(e) :$ edge e is incident on $v\}$. Fig. 4(a) pictorially depicts the coalescing of vertices v_1 and v_4 as discussed in Example 3.1. Note that, by our coalescing rule, for each vertex v, all queries in $Q(v)$ are guaranteed to use exactly the same set of attributes of $R(v)$, namely $\mathcal{A}(v)$; furthermore, by our attribute-renaming step, each query in $Q(v)$ uses each attribute in $\mathcal{A}(v)$ exactly once. This makes it possible to share an atomic sketch built for the coalesced vertices v across all queries in $Q(v)$.

Estimation with sketch sharing. Consider a multi-query join graph $\mathcal{J}(\mathcal{Q})$, possibly containing coalesced vertices (as described above), and a query $Q \in \mathcal{Q}$. Let $V(Q)$ denote the (sub)set of vertices in $\mathcal{J}(\mathcal{Q})$ attached to a join-predicate edge corresponding to Q; that is, $V(Q) = \{v :$ edge$|e$ is incident on v and $Q(e) = Q\}$. Our goal is to construct an unbiased probabilistic estimate X_Q for Q using the atomic sketches built for vertices in $V(Q)$.

The atomic sketch for a vertex v of $\mathcal{J}(\mathcal{Q})$ is constructed as follows. As before, each edge $e \in \mathcal{J}(\mathcal{Q})$ is associated with a family ξ^e of four-wise independent $\{-1, +1\}$ random variables. The difference here, however, is that edges attached to node v for the *same attribute* of $R(v)$ share the *same* ξ family; this, of course, implies that the number of *distinct* ξ families for all edges incident on v is exactly $|\mathcal{A}(v)|$ (each family corresponding to a distinct used attribute of $R(v)$). Furthermore, all distinct ξ families in $\mathcal{J}(\mathcal{Q})$ are generated independently (using mutually independent seeds). For example, in Fig. 4(a), since $A_{v_1}(e_1) = A_{v_1}(e_3) = R_1.A_1$, edges e_1 and e_3 share the same ξ family (i.e., $\xi^{e_3} = \xi^{e_1}$); on the other hand, ξ^{e_1} and ξ^{e_2} are distinct and independent. Assuming $\mathcal{A} = \{A_1, \ldots, A_k\}$ and letting $\xi^1, \ldots, \xi^k$ denote the k corresponding distinct ξ families attached to v, the atomic sketch X_v for node v is simply defined as $X_v = \sum_{(i_1,\ldots,i_k) \in A_1 \times \cdots \times A_k} f_v(i_1, \ldots, i_k) \prod_{j=1}^{k} \xi_{i_j}^j$. The final sketch-based estimate for query Q is the product of the atomic sketches over all vertices in $V(Q)$, i.e., $X_Q = \prod_{v \in V(Q)} X_v$. For instance, in Example 3.1 and Fig. 4(a), $X_{Q_1} = X_{v_1}X_{v_2}X_{v_3}$ and $X_{Q_2} = X_{v_1}X_{v_5}$.

Correctness of sketch-sharing configurations. The X_Q estimate construction described above can be viewed as simply "extracting" the join (sub)graph $\mathcal{J}(Q)$ for query Q from the multi-query graph $\mathcal{J}(\mathcal{Q})$, and constructing a sketch-based estimate for Q as described in Section 2.2. This is because, if we were to only retain in $\mathcal{J}(\mathcal{Q})$ vertices and edges associated with Q, then the resulting subgraph is identical to $\mathcal{J}(Q)$. Furthermore, our vertex coalescing (which completely determines the sketches to be shared) guarantees that Q references exactly the attributes $\mathcal{A}(v)$ of $R(v)$ for each $v \in V(Q)$, so the atomic sketch X_v can be utilized.

There is, however, an important complication that our vertex-coalescing rule still needs to address, to ensure that the atomic sketches for vertices of $\mathcal{J}(\mathcal{Q})$ provide unbiased query estimates with variance bounded as described in Theorem 2.2. Given an estimate X_Q for query Q (constructed as above), unbiasedness and the bounds on $\mathrm{Var}[X_Q]$ given in Theorem 2.2 depend crucially on the assumption that the ξ families used for the edges in $\mathcal{J}(Q)$ are distinct and independent. This means that simply coalescing vertices in $\mathcal{J}(\mathcal{Q})$ that use the same set of stream attributes is insufficient. The problem here is that the constraint that all edges for the same attribute incident on a vertex v share the same ξ family may (by transitivity) force edges for the same query Q to share identical ξ families. The following example illustrates this situation.

Example 3.2. Consider the multi-query join graph $\mathcal{J}(\mathcal{Q})$ in Fig. 3.2(b) for queries Q_1 and Q_2 in Example 3.2. ($\mathcal{J}(\mathcal{Q})$ is obtained as a result of coalescing vertex pairs v_1, v_4 and v_3, v_5 in Fig. 3.) Since $A_{v_1}(e_1) = A_{v_1}(e_3) = R_1.A_1$ and $A_{v_3}(e_2) = A_{v_3}(e_3) = R_3.A_2$, we get the constraints $\xi^{e_3} = \xi^{e_1}$ and $\xi^{e_3} = \xi^{e_2}$. By transitivity, we have $\xi^{e_1} = \xi^{e_2} = \xi^{e_3}$, i.e., all three edges of the multi-query graph share the same ξ

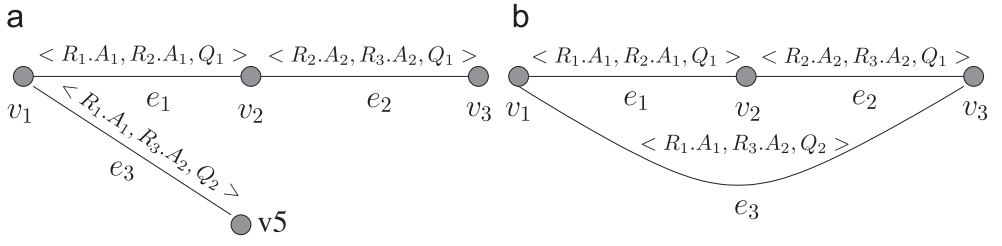

Fig. 4. Multi-query join graphs $\mathcal{J}(\mathcal{Q})$ for Example 3.1. (a) Well-formed join graph. (b) Join graph that is not well formed.

family. This, in turn, implies that the same ξ family is used on both edges of query Q_1; that is, instead of being independent, the pseudo-random families used on the two edges of Q_1 are perfectly correlated! It is not hard to see that, in this situation, the expectation and variance derivations for X_{Q_1} will fail to produce the results of Theorem 2.2, since many of the zero cross-product terms in the analysis of [8,19] will fail to vanish.

As is clear from the above example, the key problem is that constraints requiring ξ families for certain edges incident on each vertex of $\mathcal{J}(\mathcal{Q})$ to be identical, can transitively ripple through the graph forcing much larger sets of edges to share the same ξ family. We formalize this fact using the following notion of (transitive) ξ-equivalence among edges of a multi-query graph $\mathcal{J}(\mathcal{Q})$.

Definition 3.3. Two edges e_1 and e_2 in $\mathcal{J}(\mathcal{Q})$ are said to be ξ-equivalent if either (1) e_1 and e_2 are incident on a common vertex v, and $A_v(e_1) = A_v(e_2)$, or (2) there exists an edge e_3 such that e_1 and e_3 are ξ-equivalent, and e_2 and e_3 are ξ-equivalent.

Intuitively, the classes of the ξ-equivalence relation represent exactly the sets of edges in the multi-query join graph $\mathcal{J}(\mathcal{Q})$ that need to share the same ξ family; that is, for any pair of ξ-equivalent edges e_1 and e_2, it is the case that $\xi^{e_1} = \xi^{e_2}$. Since, for estimate correctness, we require that all the edges associated with a query have distinct and independent ξ families, our sketch-sharing algorithms only consider multi-query join graphs that are *well-formed*, as defined below.

Definition 3.4. A multi-query join graph $\mathcal{J}(\mathcal{Q})$ is *well-formed* iff, for every pair of ξ-equivalent edges e_1 and e_2 in $\mathcal{J}(\mathcal{Q})$, the queries containing e_1 and e_2 are distinct, i.e., $Q(e_1) \neq Q(e_2)$.

It is not hard to prove that the well-formedness condition described above is actually necessary and sufficient for individual sketch-based query estimates that are unbiased and obey the variance bounds of Theorem 2.2. Thus, our shared-sketch estimation process over well-formed multi-query graphs can readily apply the single-query results of [8,19] for each individual query in our workload.

3.2. Problem formulation

Given a large workload $\mathcal{Q}$ of complex queries, there can obviously be a large number of well-formed join graphs for $\mathcal{Q}$, and all of them can potentially be used to provide approximate sketch-based answers to queries in $\mathcal{Q}$. At the same time, since the key resource constraint in a data-streaming environment is imposed by the amount of memory available to the query processor, our objective is to compute approximate answers to queries in $\mathcal{Q}$ that are as accurate as possible given a fixed amount of memory M for the sketch synopses. Thus, in the remainder of this paper, we focus on the problem of computing (1) a well-formed join graph $\mathcal{J}(\mathcal{Q})$ for $\mathcal{Q}$, and (2) an allotment of the M units of space to the vertices of $\mathcal{J}(\mathcal{Q})$ (for maintaining iid copies of atomic sketches), such that an appropriate aggregate error metric (e.g., average or maximum error) for all queries in $\mathcal{Q}$ is minimized.

More formally, let m_v denote the sketching space (i.e., number of iid copies) allocated to vertex v (i.e., number of iid copies of X_v). Also, let M_Q denote the number of iid copies built for the query estimate X_Q. Since $X_Q = \prod_{v \in V(Q)} X_v$, it is easy to see that M_Q is actually constrained by the *minimum* number of iid atomic sketches constructed for each of the nodes in $V(Q)$; that is, $M_Q = \min_{v \in V(Q)}\{m_v\}$. By Theorem 2.2, this implies that the (square) error for query Q is equal to W_Q/M_Q, where $W_Q = 8\mathrm{Var}[X_Q]/E[X_Q]^2$ is a constant for each query Q (assuming a fixed confidence parameter δ). Our sketch-sharing optimization problem can then be formally stated as follows.

Problem statement. Given a query workload $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ and an amount of sketching memory M, compute a multi-query graph $\mathcal{J}(\mathcal{Q})$ and a space allotment $\{m_v$: for each node v in $\mathcal{J}(\mathcal{Q})\}$ such that one of the following two error metrics is minimized:

- average query error in $\mathcal{Q} = \sum_{Q \in \mathcal{Q}}(W_Q/M_Q)$
- maximum query error in $\mathcal{Q} = \max_{Q \in \mathcal{Q}}\{W_Q/M_Q\}$

subject to the constraints: (1) $\mathcal{J}(\mathcal{Q})$ is well-formed; (2) $\sum_v m_v \leqslant M$ (i.e., the space constraint is satisfied); and (3) for all vertices v in $\mathcal{J}(\mathcal{Q})$, for all queries $Q \in Q(v), M_Q \leqslant m_v$.

The above problem statement assumes that the "weight" W_Q for each query $Q \in \mathcal{Q}$ is known. Clearly, if coarse statistics in the form of histograms for the stream relations are available (e.g., based on historical information or coarse a priori knowledge of data distributions), then estimates for $E[X_Q]$ and $\mathrm{Var}[X_Q]$ (and, consequently, W_Q) can be obtained by estimating join and self-join sizes using these histograms [19]. In the event that no prior information is available, we can simply set each $W_Q = 1$; unfortunately, even for this simple case, our optimization problem is intractable (see Section 4).

In the following section, we first consider the sub-problem of optimally allocating sketching space (such that query errors are minimized) to the vertices of a *given*, well-formed join graph $\mathcal{J}(\mathcal{Q})$. Subsequently, in Section 5, we consider the general optimization problem where we also seek to determine the best well-formed multi-query graph for the given workload $\mathcal{Q}$. Since most of these questions turn out to be $\mathcal{NP}$-hard, we propose novel heuristic algorithms for determining good solutions in practice. Our algorithm for the overall problem (Section 5) is actually an iterative procedure that uses the space-allocation algorithms of Section 4 as subroutines.

4. Space allocation problem

In this section, we consider the problem of allocating space optimally given a well-formed join graph $J = \mathcal{J}(\mathcal{Q})$. We first examine the problem of minimizing the average error in Section 4.1, and then the problem of minimizing the maximum error in Section 4.2.

4.1. Minimizing the average error

We address the following more general *average-error integer convex optimization problem* in this subsection, for an arbitrary convex strictly decreasing function Φ:

$$\min \quad \sum_{Q\in\mathcal{Q}} W_Q \Phi(M_Q) \tag{1}$$

$$\forall Q \in \mathcal{Q} : M_Q > 0 \tag{2}$$

$$\forall v \in J, \forall Q \in Q(v) : M_Q \leqslant m_v \tag{3}$$

$$\sum_{v\in J} m_v = M \tag{4}$$

In the above formulation, variables M_Q and m_v correspond to the space allocated to query $Q \in \mathcal{Q}$ and vertex $v \in J$, respectively, and if we wish to minimize the average (square) error, then $\Phi(M_Q) = 1/M_Q$.

Theorem 4.1. *If Φ is convex and strictly decreasing with a singularity in 0, then solving the average-error convex optimization problem is $\mathcal{NP}$-complete. This is true even if $W_Q = 1$ for all $Q \in \mathcal{Q}$.*

Proof. We show a reduction from k-clique. A k-clique is a fully connected subgraph containing k nodes. This problem is known to be NP-hard (Garey and Johnson). Let $G = (V, E)$ be an instance of the k-clique problem. For every vertex $v \in V$ we introduce a relation R_v with a single attribute A. For every edge $e = (v_1, v_2) \in E$ we introduce a query Q_e that is the size of the join with join constraint $R_{v_1}.A = R_{v_2}.A$. Thus, the set $Q(v) = \{Q_e | e = (v, \cdot) \in E\}$. Furthermore, we set W_e the weight corresponding to query Q_e to 1 and the total memory $\mathcal{M} = n + k$ with $n = |V|$. We now show that there exists a clique of size k in G iff there exists a memory allocation strategy for the constructed problem with cost at most $B = (|E| - K)\Phi(1) + K(\Phi(1) - \Phi(2))$ where $K = k(k-1)/2$.

Since $\Phi(x)$ is ∞ in 0 we have to allocate at least one memory word for every vertex. If we have a k-clique in the graph G then by allocating the remaining k memory words the decrease in the optimization function is $K\Phi(2)$ thus the final value is B. Conversely, if we can decrease the value of the criterion from $|E|\Phi(1)$ to at least B by allocating k more memory words to k vertices it has to be the case that K edges (joins) use two memory words instead of one, thus the k edges form a k-clique. To see this observe that since $\Phi(x)$ is convex and strictly decreasing $p(\Phi(1) - \Phi(2)) > \Phi(1) - \Phi(p+1)$ so by allocating more than one extra memory word to some vertices we decrease the value of the criterion less than linearly per edge and we decrease the number of edges quadratically so it is impossible to reduce the value of the criterion by $K(\Phi(1) - \Phi(2))$ (in order to meet bound B) if we allocate more than one extra memory word for k of the vertices. □

Since this problem is $\mathcal{NP}$-hard, let us first examine its *continuous relaxation*, i.e., we allow memory to be allocated continuously although in reality we can only allocate memory in integer chunks. Thus, we allow the M_Qs and m_vs to be continuous instead of requiring them to be integers. We call this problem the *average-error continuous convex optimization problem*. In this case, we have a specialized convex optimization problem, and in the following, we show how we can use results from the theory of convex optimization to find an optimal continuous solution.[3] We then present a method to derive a near-optimal integer solution by rounding down (to integers) the optimal continuous M_Q and m_v values. In what follows, we first solve the continuous optimization problem, then we present the approximation result of the *rounded solution*.

A roadmap. Since solving the continuous optimization problem has significant technical contributions, we first outline a roadmap for the process of designing the solution. The standard technique to approach convex optimization problems is to formulate the Karush–Kuhn–Tucker (KKT) optimality conditions. The KKT conditions are a generalization of the Lagrange multiplier method to accommodate inequality constrains along with the equality constraints allowed by Lagrange multiplier method. The conditions for inequalities express the fact that either the inequality is strict and thus does not matter for the optimal solution or the inequality is satisfied by equality and the condition influences the optimum. As is the case for Lagrange multiplier method, a solution to the Lagrange conditions together with these extra conditions provide an optimal solution for the optimization problem when the objective function is convex and the region described by the conditions is convex as well. For simple problems, the fact that an inequality is important (it becomes equality) or irrelevant (strict inequality) can guessed and the KKT conditions become just regular Lagrange conditions and the optimum, quite often, can be found analytically. Unfortunately, there is no standard way to solve the KKT conditions since the solution is problem dependent. Separate strategies might have to get devised for each type of problem. This is the reason why general solvers do not solve directly the KKT conditions. Deciding which conditions are important and which are not is nontrivial for the problem at hand. A strategy that works reasonably if only a handful of inequality constraint are present is to *try* all combinations of relevant/irrelevant for each inequality constraint, solve the Lagrange problem and see if the solution satisfy the inequalities. For a large number of inequality constraints, as is the case for this problem, this is prohibitively expensive. The key idea for an efficient algorithm to find the relevant/irrelevant constraints is to introduce equivalent classes over nodes Q and v, equivalence classes that form a partitioning of the Q, v space as depicted in Fig. 5. All the inequality constraints between elements within the same class are relevant and become equality constraints. All inequality constraints between elements in two different classes become strict inequality constraints and can be ignored.

[3] We are not aware of any prior work on a specialized solution for this particular convex optimization problem. General solutions for convex optimization problems, e.g., interior point methods [23], tend to be slow in practice and would be especially problematic in this context since we need to solve such optimization problems in the inner loop of the algorithm for sketch sharing.

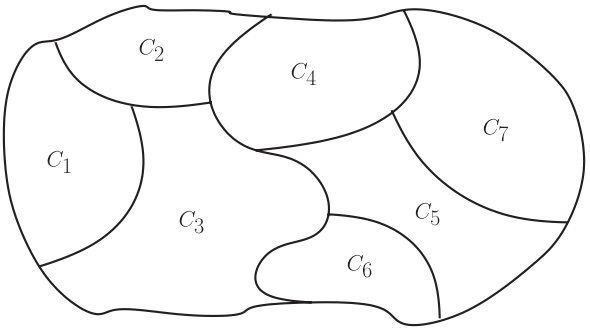

Fig. 5. Partitioning of the join graph into equivalence classes.

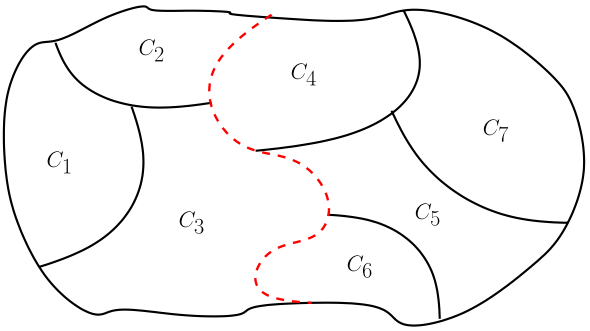

Fig. 6. Cut through the space that keeps equivalence classes intact.

Thus, if the correct partitioning in equivalent classes is determined, the problem becomes a Lagrange multiplier problem and a correct solution is readily obtained. To find the membership of the equivalence classes a number of technical results need to be derived to reveal the interactions between the classes and their elements. The final result is an procedure that uses specific max-flow problems to produce a cut through the Q, v space that does keeps the equivalence classes intact, as depicted in Fig. 6. By using this procedure, a divide-and-conquer algorithm, Fig. 7, can be devised to solve the optimization problem. Before we delve in the formal treatment of the problem, we provide a short description of the role of each technical result provided:

- KKT conditions characterize the optimal solution. Finding a solution for this conditions provides the solution.
- The $\equiv$-components are the equivalence classes over the space Q, v. These equivalence classes are the key to solve the KKT condition.
- Lemma 2 provides properties of the $\equiv$-components. Part (a) and (d) describe interactions between $\equiv$-components, part (b) provides a test to determine if a candidate set is an $\equiv$-component, and part (c) provides the connection between the optimization problem and the $\equiv$-components (i.e., how to obtain the optimal solution once the composition of the $\equiv$-components is known).
- Lemma 3 provides necessary and sufficient conditions based on a Max-Flow problem for a candidate set to be an $\equiv$-component. The lemma provides an algorithmic way to recognize $\equiv$-components.
- Lemma 4 proves that Algorithm `SelectEdges` finds cuts throughout the space like the one depicted in Fig. 6.
- Theorem 5 puts all the results together and shows that the Algorithm `ComputeSpace` correctly solves the optimization problem.

Procedure: ComputeSpace(J, M)
Require: J is a join graph, M is available memory.
Ensure: vector of m_Q's, associated error.

```
1:  V = J ∪ Q
2:  E = {(v, Q) : v ∈ J, Q ∈ Q(v)}
3:  C = ∅
4:  repeat
5:    C' = FindConnectedComponents(V, E)
6:    for all C ∈ C' do
7:      E' = SelectEdges(C)
8:      E = E \ E'
9:      if E' = ∅
10:       V = V \ C
11:       C = C ∪ C
12:     endif
13:   end for
14: until (V = ∅)
15: return ComputeMemory&Error(C, M)
```

Fig. 7. Algorithm `ComputeSpace`.

With this roadmap in mind, we can delve into the technical treatment of the problem.

The KKT conditions for convex optimization problems. The KKT conditions are necessary and sufficient optimality conditions for convex optimization problems. They are the generalization of the Lagrange conditions—the Lagrange multipliers method works only for optimization problems with equality constraints—and they allow inequality constraints as well as equality constraints. The strength of the KKT conditions is that they provide a set of equations with the property that any solution gives the optimum solution of the optimization problem—in most cases there are an infinity number of solutions of the KKT conditions even though the optimum solution is usually unique. The KKT conditions and examples of their use are provided in most *convex optimization* books—see for example [24].

Before we write the KKT conditions for our problem, we briefly review the conditions for the general case. Assume we have to solve the following general[4] optimization problem:

$$\min_{x_1,\ldots,x_n} f(x_1,\ldots,x_n)$$
$$\forall i \in \{1:k\} \quad g_i(x_1,\ldots,x_n) = 0$$
$$\forall j \in \{1:l\} \quad h_j(x_1,\ldots,x_n) \leqslant 0$$

[4] Any optimization problem can be easily written in this *canonical* way.

where $x_1,\ldots,x_n$ are the variables whose value has to be determined, $f(x_1,\ldots,x_n)$ is the optimization function that needs to be minimized under the equality constraints expressed by functions $g_1(),\ldots,g_k()$ and the inequality constraints expressed by functions $h_1(),\ldots,h_l()$. The set of points that satisfy both the equality and inequality constraints is assumed to be convex, i.e., if two points are in the set, all points on the line segment between the two points are in the set. Under these conditions, any solution of the set of equations below, called the KKT conditions, give the unique global optimum of the convex optimization problem. For a proof of this fact and more details, see for example [24]. To express the KKT conditions, a *multiplier* λ_i is introduced for each equality constraint $g_i()=0$ and another set of multipliers μ_j is introduced for each inequality constraint $h_j()\leqslant 0$. Now, if we define function $\mathscr{L}$, called the Lagrangian, as:

$$\mathscr{L}(x_1,\ldots,\lambda_1,\ldots,\mu_1,\ldots)=f(x_1,\ldots,x_n)-\sum_{i=1}^{k}\lambda_k g_i(x_1,\ldots)-\sum_{j=1}^{l}\lambda_k h_j(x_1,\ldots)$$

then the KKT conditions are

$$\begin{aligned}&\forall i\in\{1:n\}\quad \frac{\partial\mathscr{L}}{\partial x_i}=0\\ &\forall i\in\{1:k\}\quad \lambda_i>0,\ g_i(x_1,\ldots)=0\\ &\forall j\in\{1:l\}\quad \mu_j h_j(x_1,\ldots)=0,\\ &\mu_j\geqslant 0,\ \ h_j(x_1,\ldots)\leqslant 0\end{aligned}$$

The first two conditions come from the Lagrange multiplier method but the third set of conditions are specific to KKT method only. The condition $\mu_j h_j(x_1,\ldots)=0$ expresses the fact that either $\mu_j=0$, case in which the condition does not influence the optimum, or $h_j(x_1,\ldots)=0$, in which case the inequality is satisfied with equality. As we mentioned above, if a correct *guess* is made which multipliers μ_j have to be 0 and which not, the problem becomes a regular Lagrange multiplier problem. The difficulty is finding a correct *guess* for which the optimum of the resulting Lagrange multiplier problem satisfies all the conditions $h_j(x_1,\ldots)\leqslant 0$. Thus, a specialized solution of the KKT conditions consists in the guess of the zero multipliers and solving the resulting Lagrange multipliers problem. There is no general method to efficiently *guess* the zero multipliers; the structure of the problem needs to be exploited to find efficient methods for the guess.

To formulate the KKT conditions for the problem at hand, we first observe that if we set $M_Q=m_v=M/|J|$, we have a solution to the average-error continuous optimization problem; this solution may not be optimal, but it satisfies Eqs. (2)–(4). In addition, since Φ is strictly convex and the set of feasible solutions is convex, the problem has a single global optimum which we refer to as the *optimal solution*.

We can characterize the optimal solution completely through the KKT conditions [24]. The Lagrangian has the following form:

$$L(\mu_{v,Q},\lambda)=\sum_{Q\in\mathscr{Q}}W_Q\Phi(M_Q)-\sum_{v\in J}\sum_{Q\in Q(v)}\mu_{v,Q}(m_v-M_Q)-\lambda\Big(M-\sum m_v\Big)$$

This results in the following set of KKT conditions:

$$\begin{aligned}&\forall Q\in\mathscr{Q}:\quad W_Q\Phi'(M_Q)+\sum_{v\in V(Q)}\mu_{v,Q}=0\\ &\quad\Big(\text{solve}\ \frac{\partial L}{\partial M_Q}=0\Big)\\ &\forall v\in J:\quad -\sum_{Q\in Q(v)}\mu_{v,Q}+\lambda=0\quad\Big(\text{solve}\frac{\partial L}{\partial m_v}=0\Big)\\ &\forall Q\in\mathscr{Q},\forall v\in J:\quad \mu_{v,Q}\cdot(m_v-M_Q)=0\\ &\sum_{v\in J}m_v=M\\ &\forall Q\in\mathscr{Q},\forall v\in J:\quad \mu_{v,Q}\geqslant 0,\quad \lambda>0\end{aligned}$$

Since $\lambda>0$ we can rewrite the KKT conditions as follows (substituting $\bar{\mu}_{v,Q}$ for $\mu_{v,Q}/\lambda$):

$$\forall Q\in\mathscr{Q}:\quad -W_Q\Phi'(M_Q)=\lambda\cdot\sum_{v\in V(Q)}\bar{\mu}_{v,Q}\tag{5}$$

$$\forall v\in J:\quad \sum_{Q\in Q(v)}\bar{\mu}_{v,Q}=1\tag{6}$$

$$\forall Q\in\mathscr{Q},\forall v\in J:\quad \bar{\mu}_{v,Q}\cdot(m_v-M_Q)=0\tag{7}$$

$$\sum_{v\in J}m_v=M\tag{8}$$

$$\forall Q\in\mathscr{Q},\forall v\in J:\quad \bar{\mu}_{v,Q}\geqslant 0\tag{9}$$

Note that the above KKT conditions are necessary and sufficient, that is, a solution for our continuous convex optimization problem is optimal if and only if it satisfies the KKT conditions.

Characterizing the optimal solution. The KKT conditions enable us to identify structural properties of the optimal solution. Let us first introduce some notation. A *component* C is a subset of $\mathscr{Q}\cup J$. For a component C, define $V(C)=\{v:v\in(C\cap J)\}$, $Q(C)=\{Q:Q\in(C\cap\mathscr{Q})\}$, $E(C)=\{(v,Q):v\in V(C),Q\in Q(C)\cap Q(v)\}$, and define $W(C)=\sum_{Q\in Q(C)}W_Q$. We consider a special set of components determined by the optimal solution that we call $\equiv$-components. We define a relation $\equiv$ between $v\in J$ and $Q\in\mathscr{Q}$ as follows: $v\equiv Q$ iff $(m_v=M_Q\wedge Q\in Q(v))$ in the optimal solution. If we take the symmetric transitive closure of $\equiv$ we obtain an equivalence relation that partitions $J\cup\mathscr{Q}$ into a set of components $\mathscr{C}=\{C_1,\ldots,C_c\}$ which we call $\equiv$-components. Each $\equiv$-component $C\in\mathscr{C}$ has an associated memory allocation $M(C)$, i.e., since C is a $\equiv$-component, $\forall v\in V(C),m_v=M(C)$ and $\forall Q\in Q(C),M_Q=M(C)$.

Lemma 2. *The set $\mathscr{C}$ of $\equiv$-components has the following properties*:

(a) *Let $C,C'\in\mathscr{C}$ and $C\neq C'$. Then $\forall v\in V(C),\forall Q\in Q(C')$, it is the case that $\bar{\mu}_{v,Q}=0\wedge M(C)\geqslant M(C')$.*

(b) *For any* $\equiv$*-component* C,

$$\forall Q \in Q(C): \quad W_Q \cdot \frac{|V(C)|}{W(C)} = \sum_{v \in V(C)} \bar{\mu}_{v,Q} \tag{10}$$

$$\forall v \in V(C): \quad \sum_{Q \in Q(v) \cap Q(C)} \bar{\mu}_{v,Q} = 1 \tag{11}$$

(c) *The memory allocation for the* $\equiv$*-components satisfies the following two equations*:

$$\forall C \in \mathscr{C}: \quad -W(C)\Phi'(M(C)) = \lambda |V(C)| \tag{12}$$

$$\sum_{C \in \mathscr{C}} M(C) \cdot |V(C)| = M \tag{13}$$

(d) $\forall C, C' \in \mathscr{C} : M(C) < M(C')$ *iff* $W(C)/|V(C)| < W(C')/|V(C')|$

Proof. (a) Suppose that, for $v \in V(C)$ and $Q \in Q(C')$, Eqs. (5)–(9) have a solution with $\bar{\mu}_{v,Q} > 0$. Eq. (7) implies that $m_v = M_Q$ which in turn implies that $v \equiv Q$, and as a result, $C = C'$ by the definition of a $\equiv$-component. This leads to a contradiction, and thus $\bar{\mu}_{v,Q} = 0$. Further, suppose that $M(C) < M(C')$, which implies that $m_v < M_Q$. However, we again have a contradiction because $Q \in Q(v)$, and as a result, $M_Q \leqslant m_v$ in the optimal solution (due to Eq. (3)).

(b) Consider a $\equiv$-component C. Due to part (a), we can rewrite Eqs. (5) and (6) as follows:

$$\forall Q \in Q(C): \quad -W_Q \frac{\Phi'(M(C))}{\lambda} = \sum_{v \in V(Q) \cap C} \bar{\mu}_{v,Q} \tag{14}$$

$$\forall v \in V(C): \quad \sum_{Q \in Q(v) \cap C} \bar{\mu}_{v,Q} = 1 \tag{15}$$

If we sum Eq. (14) over all $Q \in Q(C)$, and we sum Eq. (15) over all $v \in V(C)$, we obtain the following two equations:

$$-W(C) \frac{\Phi'(M(C))}{\lambda} = \sum_{Q \in Q(C)} \sum_{v \in V(Q) \cap C} \bar{\mu}_{v,Q} \tag{16}$$

$$\sum_{v \in V(C)} \sum_{Q \in Q(v) \cap C} \bar{\mu}_{v,Q} = |V(C)| \tag{17}$$

Now we immediately have $-\Phi'(M(C))/\lambda = |V(C)|/W(C)$, which when substituted into Eqs. (14) and (15) completes the proof.

(c) For a given $C \in \mathscr{C}$, we sum Eq. (5) over all $Q \in Q(C)$. We then use the result from Eq. (17) and obtain Eq. (12). Since for a $\equiv$-component C we know that $\forall v \in V(C)$, $m_v = M(C)$, Eq. (8) can be rewritten as Equation (13).

(d) First, note that $-1/\Phi'(M(C))$ is proportional to $W(C)/|V(C)|$ by Eq. (12). Since function Φ' is negative and strictly decreasing, $-1/\Phi'$ is positive and strictly increasing. Thus $M(C) < M(C')$ implies $-1/\Phi'(M(C)) < -1/\Phi'(M(C'))$ which gives the result. □

Once we have identified the set $\mathscr{C}$ of $\equiv$-components (for the optimal solution), then part (c) of Lemma 2 tells us how to allocate memory optimally to the various components $C \in \mathscr{C}$, and also, its vertices and queries since $m_v = M_Q = M(C)$ for all $v \in V(C)$ and $Q \in Q(C)$. Thus, we simply need to compute the set $\mathscr{C}$, and parts (a), (b), and (d) of Lemma 2 guide us in identifying this set. We now give a result that allows us to efficiently check whether Part (b) in Lemma 2 is true for a candidate $\equiv$-component C; this result is Lemma 3. Note that the other parts can be checked in a straightforward way. Before we state Lemma 3, let us introduce some notation.

Let us define the *flow graph* F of a component C as follows: $F(C)$ is a directed graph with capacities on the edges. The vertices of $F(C)$ are the elements of C plus two designated vertices s and t. $F(C)$ contains the following edges: (1) $\forall v \in V(C)$, edge (s, v) with capacity 1, (2) $\forall v \in V(C), \forall Q \in Q(v) \cap Q(C)$, edge (v, Q) with capacity ∞, and (3) $\forall Q \in Q(C)$, edge (Q, t) with capacity $W_Q \cdot |V(C)|/W(C)$.

Now, a flow from s to t assigns a positive real value to each edge in $F(C)$ such that (1) the flow value for each edge does not exceed the edge's capacity, and (2) for each vertex, the flow is conserved, that is, the sum of the flows along incoming edges is equal to the sum of the flows along outgoing edges. The maximum flow is one for which the flow out of s is maximum (note that due to flow conservation, this is also the flow into t). Given a flow, we refer to a vertex $v \in V(C)$ as *saturated* if the flow entering v is equal to the capacity of edge (s, v), which is 1. Similarly, vertex $Q \in Q(C)$ is said to be saturated if the flow out of Q equals the capacity of (Q, t), which is $W_Q \cdot |V(C)|/W(C)$. We call vertices that are not saturated as simply *unsaturated* vertices.

Lemma 3. *Let* C *be a component, and let* $F(C)$ *be its flow graph. Eqs.* (10) *and* (11) *have a solution if and only if there is a flow between* s *and* t *of size* $|V(C)|$.

Proof. First, observe that in order the flow from s to t to have size $|V(C)|$, all vertices $v \in V(C)$ and $Q \in Q(C)$ have to be saturated. Since Eqs. (10) and (11) are exactly the flow conservation equations whenever the flow from s to t is exactly $|V(C)|$, the equations have a solution if and only if the maximum flow from s to t is $|V(C)|$. □

Note that we can implement the check in Lemma 3 efficiently with any max-flow algorithm, for example the Ford–Fulkerson Algorithm [25].

Algorithm for finding optimal solution. We are now in a position to present our algorithm for determining the set $\mathscr{C}$ of $\equiv$-components that characterize the optimal solution, and which can then be used to compute the optimal values for M_Q and m_v using Lemma 2(c). At a very high level, our optimal space computation procedure (see Fig. 7) starts with the initial component $C = J \cup \mathscr{Q}$ and the set of edges $E = E(C)$. In each iteration of the outermost loop, it deletes from E a subset E' of edges between pairs of distinct $\equiv$-components, until the final set of $\equiv$-components is extracted into $\mathscr{C}$. The procedure `SelectEdges` computes the set of edges E' deleted in each iteration, and in the final step, $\mathscr{C}$ is used to compute the optimal memory allocation M_Q (by solving Eqs. (12) and (13)) that minimizes the error $\sum_Q W_Q \Phi(M_Q)$.

We now turn our focus to Algorithm `SelectEdges`, which uses Lemmas 2 and 3, to identify the edges

between $\equiv$-components in C that should be deleted. In the algorithm, MF is a max-flow solution of the flow graph $F(C)$ of C, and S_F and S_B are the sets of vertices reachable in C from *unsaturated* v and Q vertices, respectively. Note that when computing the vertex set $S_F(S_Q)$, an edge (v,Q) is traversed in the direction from Q to v (v to Q) only if $MF(v,Q)>0$. In Lemma 4 below, we shown that all edges returned by `SelectEdges` in Step 4 are between $\equiv$-components.

Let $C_1,\ldots,C_l$ be the $\equiv$-components in C. Note that C is connected with respect to $E(C)$, and since, as we show below, `SelectEdges` only returns edges between $\equiv$-components, C contains only entire $\equiv$-components. Let us define two different sets $\mathscr{T}_<$ and $\mathscr{T}_>$ of $\equiv$-components that contain all $\equiv$-components C_i in C for which $(W(C_i)/|V(C_i)|)<(W(C)/|V(C)|)$, and $(W(C_i)/|V(C_i)|)>(W(C)/|V(C)|)$, respectively. Further, let $E'=\{(v,Q): v\in V(C)\wedge Q\in Q(C)\cap Q(v)\wedge((Q\in\mathscr{T}_<\wedge v\notin\mathscr{T}_<)\vee(Q\notin\mathscr{T}_>\wedge v\in\mathscr{T}_>))\}$.

We now present Fig. 7 that computes a solution to the KKT conditions. To prove that the algorithm is correct, we now prove in Lemma 4 a property about Fig. 8, a subroutine of Fig. 7. Recall that $\mathscr{C}=C_1,\ldots,C_c$ is the set of $\equiv$-components of the optimal solution.

Lemma 4. *Consider an invocation of Algorithm* `Select-Edges` *with component C. Then the following properties hold.*

(a) *If $E'=\emptyset$, then C contains exactly one $\equiv$-component.*
(b) *The following is true for sets S_F and S_B computed in the body of the algorithm.*

$$\bigcup_{C_i\in\mathscr{T}_<} C_i = S_F \tag{18}$$

$$\bigcup_{C_i\in\mathscr{T}_>} C_i = S_B \tag{19}$$

(c) *Algorithm* `SelectEdges` *returns exactly the set E'.*

Proof. (a) If $E'=\emptyset$, then for every $\equiv$-component C_i in C, $W(C_i)/|V(C_i)|=W(C)/V(C)$. Thus, due to Lemma 2(c), all $M(C_i)$ are equal in the optimal solution, and since C is connected with respect to $E(C)$, it follows that all vertices in C belong to a single $\equiv$-component.

(b) We prove Eq. (18) in part (b) in four steps. (The proof of Eq. (19) is similar.) In the following, we use the symbols v and Q generically to refer to vertices in $V(C)$ and $Q(C)$, respectively.

Step 1: There cannot be an edge (v,Q) in the flow graph $F(C)$ such that $v\in\mathscr{T}_<$ and $Q\notin\mathscr{T}_<$. This is because, due to Lemma 2(d), we would get $m_v<M_Q$ in the optimal solution, which is not feasible.

Procedure: SelectEdges(C)
Require: C is a component.
Ensure: Returns the set of edges to be deleted from C.
1: MF = Maxflow($F(C)$)
2: S_F = Forward-Mark(C, MF)
3: S_B = Backward-Mark(C, MF)
4: return $\{(v,Q): v\in V(C)\wedge Q\in Q(C)\cap Q(v)\wedge((Q\in S_F\wedge v\notin S_F)\vee(Q\notin S_B\wedge v\in S_B))\}$

Fig. 8. Algorithm `SelectEdges`.

Step 2: All $Q\in\mathscr{T}_<$ and all $v\notin\mathscr{T}_<$ are saturated, and for every edge (v,Q) in $F(C)$ such that $v\notin\mathscr{T}_<$ and $Q\in\mathscr{T}_<, MF(v,Q)=0$ in the max-flow solution. Consider any component C_i in $\mathscr{T}_<$. We know that $(W(C_i)/|V(C_i)|)<(W(C)/|V(C)|)$, and so for each Q vertex in C_i, the capacity $W_Q(|V(C)|/W(C))$ of edges (Q,t) in $F(C)$ is less than $W_Q(|V(C_i)|/W(C_i))$. Note that from Lemma 3, we know that with (Q,t) edge capacities set to $W_Q(|V(C_i)|/W(C_i))$, all Q vertices in $\mathscr{T}_<$ can be saturated with the incoming flow into $\mathscr{T}_<$. Thus, due to Step 1 above, since there is no flow out of $\mathscr{T}_<$, (with smaller (Q,t) edge capacities) we get that in MF all Q vertices in $\mathscr{T}_<$ are saturated, and $\mathscr{T}_<$ contains at least one unsaturated v vertex. By a symmetric argument, for every C_i not in $\mathscr{T}_<$, since $(W(C_i)/|V(C_i)|)\geqslant(W(C)/|V(C)|)$, we can show that all the incoming flow into vertices $v\notin\mathscr{T}_<$ can be pushed out of Q vertices not in $\mathscr{T}_<$. Thus, in the max-flow solution MF, there cannot be any flow along edge (v,Q), where$v\notin\mathscr{T}_<$ and $Q\in\mathscr{T}_<$, since pushing any such flow out of a Q vertex not in $\mathscr{T}_<$ would increase the total flow from s to t beyond MF. Thus, it follows that in MF, all v vertices not in $\mathscr{T}_<$ are saturated.

Step 3: Now let us consider the set S_F computed by `Forward-Mark` (Figs. 9 and 10). Clearly, since only $\mathscr{T}_<$ contains unsaturated v vertices, there are no out-edges from a vertex $v\in\mathscr{T}_<$ (due to Step 1), and incoming edges into a vertex $Q\in\mathscr{T}_<$ have a flow of 0 (due to Step 2), vertices that do not belong to $\mathscr{T}_<$ will not be added to S_F. Thus, we only need to show that all the vertices in $\mathscr{T}_<$ will be added to S_F. We do this in the next step.

Step 4: Suppose that S'_F is the subset of v and Q vertices in $T_<$ that do not belong to S_F. Clearly, all the v vertices in S'_F must be saturated (since otherwise, they would have

Procedure: Forward-Mark(C, MF)
Require: C is a component, MF is a max-flow solution of $F(C)$.
Ensure: All vertices in components from $\mathcal{T}_<$.
1: $S=\{v: v\in V(C)\wedge MF(s,v)<1\}$
2: **repeat**
3: $S'=S$
4: $S=S\cup\{v: Q\in S\wedge v\in V(C)\cap V(Q)\wedge MF(v,Q)>0\}$
5: $S=S\cup\{Q: v\in S\wedge Q\in Q(C)\cap Q(v)\}$
6: **until** $(S'=S)$
7: return S

Fig. 9. Algorithm `Forward-Mark`.

Procedure: Backward-Mark(C, MF)
Require: C is a component, MF is a max-flow solution of $F(C)$.
Ensure: All vertices in components from $\mathcal{T}_>$.
1: $S=\{Q: Q\in Q(C)\wedge MF(Q,t)<W_Q\frac{|V(C)|}{W(C)}\}$
2: **repeat**
3: $S'=S$
4: $S=S\cup\{v: Q\in S\wedge v\in V(C)\cap V(Q)\}$
5: $S=S\cup\{Q: v\in S\wedge Q\in Q(C)\cap Q(v)\wedge MF(v,Q)>0\}$
6: **until** $(S'=S)$
7: return S

Fig. 10. Algorithm `Backward-Mark`.

been added to S_F). Similarly, we can show that there is no flow out of S'_F in the max-flow *MF*. We show that there must be an edge into a Q vertex in S'_F from a v vertex in S_F; but this would cause the Q vertex to be added to S_F, and thus lead to a contradiction. Suppose that there is no (v, Q) edge from S_F to S'_F. Then, this would imply that if the capacity of each (Q, t) edge for $Q \in S'_F$ were increased, it would not be possible to saturate all the Q vertices in S'_F with the incoming flow into $\mathscr{T}_<$, and this violates Lemma 3. The reason for not being able to saturate all Q vertices in S'_F is that every v vertex in S'_F is already saturated, there is no outgoing flow from S'_F in *MF*, and there are no incoming edges into S'_F from S_F. Thus, S_F contains all vertices in $\mathscr{T}_<$.

(c) Part (c) follows directly from Part (b) above and the set of (v, Q) edges returned in Step 4 of `SelectEdges`. □

Theorem 5. *Algorithm* `ComputeSpace` *computes the optimal solution to the average-error continuous convex optimization problem in at most* $O(\min\{|\mathscr{Q}|, |J|\} \cdot (|\mathscr{Q}| + |J|)^3)$ *steps.*

Proof. There can be at most $\min\{|\mathscr{Q}|, |J|\}$ $\equiv$-components in $\mathscr{C}$, and by Lemma 4(c), each call to `SelectEdges` with component C causes the $\equiv$-components in $\mathscr{T}_<$ and $\mathscr{T}_>$ to become disconnected. Thus, `SelectEdges` is invoked at most $2\min\{|\mathscr{Q}|, |J|\}$ times, and since the time complexity of each invocation is dominated by $O((|\mathscr{Q}| + |J|)^3)$, the number of steps required to compute the max-flow for components containing at most $|\mathscr{Q}| + |J|$ vertices, the time complexity of `ComputeSpace` is $O(\min\{|\mathscr{Q}|, |J|\} \cdot (|\mathscr{Q}| + |J|)^3)$. By Lemma 4(a), `ComputeSpace` terminates only if $\mathscr{C}$ contains individual $\equiv$-components. Thus, solving the equations in Lemma 2(c), we can compute the optimal solution and its error. □

In the following example, we trace the execution of `ComputeSpace` for a join graph J.

Example 6. Consider a join graph J with vertices $v_1, \ldots, v_5$. Let $\mathscr{Q} = \{Q_1, Q_2, Q_3\}$ and let $V(Q_1) = \{v_1, v_2, v_4\}$, $V(Q_2) = \{v_2, v_3\}$ and $V(Q_3) = \{v_4, v_5\}$. Also, let $W_{Q_1} = W_{Q_2} = 3$ and $W_{Q_3} = 9$. The flow graph $F(C)$ for the initial connected component C with which `SelectEdges` invoked (in the first iteration of `ComputeSpace`) is depicted in Fig. 11(a). Each edge in the figure is labeled with its capacity and the max-flow that can be pushed along the edge. For instance, the capacity for the edge out of Q_1 is $W_{Q_1}(|V(C)|/W(C)) = 3\frac{5}{15} = 1$, whereas the capacity for the outgoing edge from Q_3 is equal to $W_{Q_3}(|V(C)|/W(C)) = 9\frac{5}{15} = 3$. Also, all vertices except for v_3 and Q_3 are saturated. Further, the sets $S_F = \{v_1, v_2, v_3, Q_1, Q_2\}$ (reachable from v_3, but not traversing 0-flow edges from a Q vertex to a v vertex) and $S_B = \{v_4, v_5, Q_3\}$ (reachable from Q_3, but not traversing 0-flow edges from a v vertex to a Q vertex). Thus, since $Q_1 \in S_F$ and $v_4 \in S_B$, edge $(v4, Q_1)$ is returned by `SelectEdges` and deleted from the edge set E. In the second iteration, `ComputeSpace` invokes `SelectEdges` with the following two connected components: $C_1 = \{v_1, v_2, v_3, Q_1, Q_2\}$ and $C_2 = \{v_4, v_5, Q_3\}$. The edge capacities and max-flows for each component is shown in Fig. 11(b). For instance, the capacity for the edge out of Q_1 is $W_{Q_1}(|V(C_1)|/W(C_1)) = 3\frac{3}{6} = 3/2$, whereas the

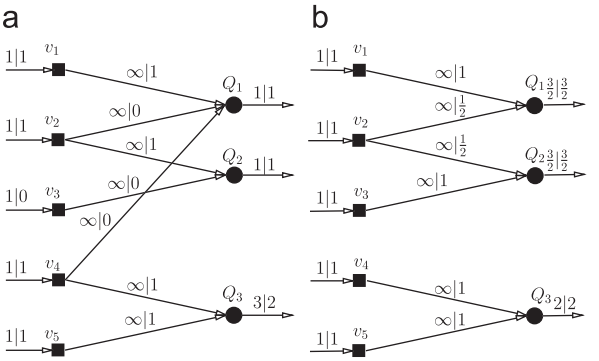

Fig. 11. Example trace of Algorithm `ComputeSpace`. (a) Iteration 1. (b) Iteration 2.

capacity for the outgoing edge from Q_3 is equal to $W_{Q_3}(|V(C_2)|/W(C_2)) = 9\frac{2}{9} = 2$. Since there are no unsaturated vertices, $S_F = S_B = \emptyset$ and `SelectEdges` returns no edges, thus causing `ComputeSpace` to terminate and return the space allocation for $\mathscr{C} = \{C_1, C_2\}$. Solving Eqs. (12) and (13), we get $M(Q_1) = M(Q_2) = m_{v_1} = m_{v_2} = m_{v_3} = M(C_2) = M/6$ and $M(Q_3) = m_{v_4} = m_{v_5} = M(C_2) = M/4$.

The final remaining step is to go from the optimal continuous solution to a near-optimal integer solution, by rounding down each M_Q returned by Algorithm `ComputeSpace`. Clearly, by rounding down each M_Q to the biggest integer less than or equal to M_Q, our near-optimal solution still satisfies Eqs. (2)–(4).[5] In addition, we can show that the average-error for the rounded down solution is not too far from the average-error for the optimal integral solution.

Theorem 7. *The average-error of the rounded optimal continuous solution is no more than* $(1 + 2|J|/M)$ *times the average-error of the optimal integral solution.* (*Note that for optimizing average-error, we choose* $\Phi(M_Q) = 1/M_Q$.)

Proof. Suppose that $\mathscr{C} = \{C_1, \ldots, C_c\}$ is the set of $\equiv$-components. Then, solving Eqs. (12) and (13), we get that each

$$M(C_i) = \frac{M}{\sum_j \sqrt{W(C_j) \cdot |V(C_j)|}} \sqrt{\frac{W(C_i)}{|V(C_i)|}}$$

Thus, the average error for the continuous optimal solution is given by

$$\sum_Q \frac{W_Q}{M_Q} = \sum_j \frac{W(C_j)}{M(C_j)} = \frac{(\sum_j \sqrt{W(C_j) \cdot |V(C_j)|})^2}{M}$$

[5] If $M_Q < 1$, then we can avoid M_Q from being rounded down to 0 by pre-allocating 1 unit of memory to every $Q \in \mathscr{Q}$ and $v \in J$. Thus, we would then choose $\Phi(M_Q)$ to be $1/(1 + M_Q)$ instead of $1/M_Q$, and the available memory to be $M - |J|$.

Now, the error for the rounded down solution is $\sum_Q W_Q/\lfloor M_Q \rfloor = \sum_j W(C_j)/\lfloor M(C_j) \rfloor$. Since

$$\frac{1}{\lfloor M(C_j) \rfloor} \leqslant \frac{2 + M(C_j)}{M(C_j)^2}$$

we can derive the following (after substituting for $M(C_j)$):

$$\sum_Q \frac{W_Q}{\lfloor M_Q \rfloor} \leqslant \left(1 + \frac{2|J|}{M}\right) \sum_Q \frac{W_Q}{M_Q}$$

Thus, the theorem follows since the average error for the optimal continuous solution cannot be more than the average error for the optimal integral solution. □

4.2. Minimizing the maximum error

We now turn our attention to the problem of allocating space to the vertices of J to minimize the maximum query error; that is, we seek to minimize the quantity $\max_{Q \in \mathcal{Q}} \{W_Q/M_Q\}$, subject to the constraints: (1) $\sum_v m_v \leqslant M$ and (2) $M_Q = \min_{v \in V(Q)}\{m_v\}$. Fortunately, this turns out to be a much simpler problem than the average-error case—we can actually solve it optimally using a simple algorithm that relies on fairly standard discrete-optimization techniques [24].

To see this, we first perform a simple transformation of our objective to obtain an equivalent max–min problem. Clearly, our problem is equivalent to *maximizing* $\min_{Q \in \mathcal{Q}}\{M_Q/W_Q\}$ subject to the same constraints for M_Q, m_v. Since, $M_Q = \min_{v \in V(Q)}\{m_v\}$, some simple rewriting of the objective function gives

$$\begin{aligned}\min_{Q \in \mathcal{Q}}\left\{\frac{M_Q}{W_Q}\right\} &= \min_{Q \in \mathcal{Q}}\left\{\frac{\min_{v \in V(Q)}\{m_v\}}{W_Q}\right\} \\ &= \min_v\left\{m_v \min_{Q \in Q(v)} \frac{1}{W_Q}\right\} \\ &= \min_v\left\{\frac{m_v}{\max_{Q \in Q(v)} W_Q}\right\}\end{aligned}$$

Since $\max_{Q \in Q(v)} W_Q$ is a constant for a given vertex v, the above transformation shows that our maximum-error problem is basically equivalent to a linear max–min optimization which can be solved optimally using standard techniques [24]. A simple (optimal) algorithm is to first compute the optimal continuous solution (where each m_v is simply proportional to $\max_{Q \in Q(v)} W_Q$), round down each m_v component to the nearest integer, and then take the remaining space $s \leqslant |J|$ and allocate one extra unit of space to each of the nodes with the s smallest values for $m_v/\max_{Q \in Q(v)} W_Q$. The complexity of this procedure is $O(|J| \log |J|)$ and a proof of its optimality can be found in [24].

5. Computing a well-formed join graph

In the previous section, we showed that for a *given* well-formed join graph $\mathcal{J}(\mathcal{Q})$, computing the optimal space allocation to the vertices of $\mathcal{J}(\mathcal{Q})$ such that the average error is minimized, is an $\mathcal{NP}$-hard problem (Theorem 4.1). The optimization problem we are interested in solving is actually more general, and involves finding a join graph that is both well-formed and for which the query error is minimum. Unfortunately, this problem is tougher than the space allocation problem that we tackled in the previous section, and is thus $\mathcal{NP}$-hard for the average error case. Further, even though we optimally solved the space allocation problem for the maximum error case (see previous section), the joint problem of finding a well-formed graph for which the maximum query error is minimized, is $\mathcal{NP}$-hard. In fact, even for the simple case when $W_Q = 1$ for all queries, the joint problem is $\mathcal{NP}$-hard. The reason for this is that when all queries have the same weight, then the maximum error is minimized when M_Q for all queries in $\mathcal{J}(\mathcal{Q})$ are equal. This implies that, in the optimal solution, the memory M is distributed equally among vertices of the join graph, and the joint problem reduces to that of finding a well-formed join graph $\mathcal{J}(\mathcal{Q})$ with the minimum number of vertices—this problem is $\mathcal{NP}$-hard due to the following theorem.

Theorem 8. *The problem of finding a well-formed join graph $\mathcal{J}(\mathcal{Q})$ with the minimum number of vertices is $\mathcal{NP}$-complete.*

Proof. We show a reduction from the vertex cover problem, an instance of which seeks to find vertex cover of size k for a given graph $G = (V, E)$. For an instance of the vertex cover problem, we construct an instance of our problem of finding the smallest well-formed join graph in $\mathcal{J}(\mathcal{Q})$ as follows. For each vertex $v \in V$, there is a relation R_v, and for every edge $e = (u, v)$ in E, there are three relations $R_e, R_{e,u}$, and $R_{e,v}$. Our query set $\mathcal{Q}$ contains the following three queries per edge $e = (u, v)$ in E: $Q_e =$ `SELECT COUNT FROM` R_u, R_v `WHERE` $R_u.A_1 = R_v.A_1$, $Q_{e,u} =$ `SELECT COUNT FROM` $R_u, R_e, R_{e,u}$ `WHERE` $R_u.A_1 = R_e.A_2 \wedge R_e.A_3 = R_{e,u}.A_3$, and $Q_{e,v} =$ `SELECT COUNT FROM` $R_v, R_e, R_{e,v}$ `WHERE` $R_v.A_1 = R_e.A_3 \wedge R_e.A_2 = R_{e,v}.A_2$. Fig. 12(a) depicts the join subgraph for the three queries $Q_e, Q_{e,u}$, and $Q_{e,v}$ corresponding to edge $e = (u, v)$. In the figure, all vertices for the same relation are coalesced in the join graphs $\mathcal{J}(Q_e), \mathcal{J}(Q_{e,u})$ and $\mathcal{J}(Q_{e,v})$, and each vertex is labeled with its corresponding relation. Each edge is labeled with its corresponding triple, and edges for different queries are represented using different types of lines. Observe that for an edge $e = (u, v) \in E$, relation R_e only appears in queries $Q_{e,u}$ and $Q_{e,v}$, and for a vertex $v \in V$, relation R_v appears in queries Q_e and $Q_{e,v}$ for every edge e incident on v in G.

The key observation we make is that the join subgraph for edge e in Fig. 12(a) is not well formed. The reason for this is that due to the common attributes $R_u.A_1$, $R_v.A_1, R_e.A_2$ and $R_e.A_3$, all edges are forced to share the same ξ family. Consequently, the ξ families for the edges belonging to queries $Q_{e,u}$ and $Q_{e,v}$ are identical. Now consider one of the relations R_u or R_v, say R_v. Suppose we do not coalesce the vertices for R_v in $\mathcal{J}(Q_e)$ and $\mathcal{J}(Q_{e,v})$, causing the resulting join subgraph for e to be well-formed, as shown in Fig. 12(b). The reason for this is that

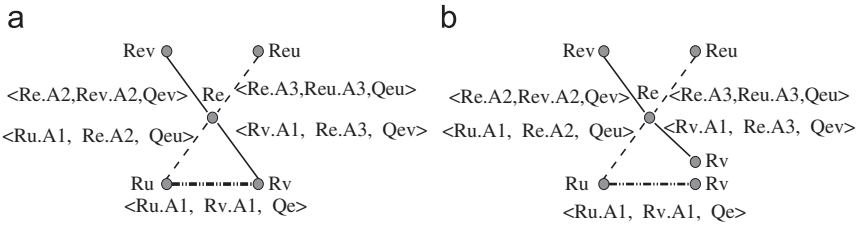

Fig. 12. Join subgraph for edge e in G.

only the ξ family for the edge pair incident on R_e and associated with attribute $R_e.A_2$ (or alternately, $R_e.A_3$) are forced to be the same, and these belong different queries $Q_{e,u}$ and $Q_{e,v}$. In the following, we show that G has a vertex cover of size k if and only if there exists a well-formed join graph containing no more than $|V| + 3|E| + k$ vertices.

Suppose that V' is a vertex cover for G of size k. Then we construct the well-formed join graph J by coalescing vertices in $\mathcal{J}(Q_e)$, $\mathcal{J}(Q_{e,u})$ and $\mathcal{J}(Q_{e,v})$ for all $e \in E$ as follows. For each edge $e \in E$, coalesce all vertices for relation R_e in J, and for every vertex $v \notin V'$, coalesce all vertices for relation R_v in J. For each vertex $v \in V'$, coalesce all vertices for R_v and belonging to queries Q_e into one vertex, and coalesce the remaining vertices for R_v (and belonging to queries $Q_{e,v}$) into a separate vertex. Since for each edge $e = (u, v) \in E$, one of u or v is in V' (say v), the resulting join subgraph for edge e in J is as shown in Fig. 12(b). Thus, as we argued earlier, edges for queries $Q_{e,u}$ and $Q_{e,v}$ are not forced to share the same ξ family. Also, J contains at most $|V| + 3|E| + k$ vertices: $3|E|$ vertices for relations $R_e, R_{e,u}$, and $R_{e,v}$, $|V| - k$ vertices for relations $R_v, v \in V - V'$, and $2k$ vertices for relations $R_v, v \in V'$.

On the other hand, suppose there exists a well formed join graph J containing no more than $|V| + 3|E| + k$ vertices. Then, clearly, for each $e = (u, v)$ in E, there must be two vertices for one of R_v or R_u since otherwise J would contain the subjoin graph in Fig. 12(b), and thus cannot be well formed (note that while it is possible that J contains two vertices for R_e, the same effect can be achieved by two vertices for R_u or R_v). Thus, if we define V' to be the set of vertices in V such that J contains more than one vertex for R_v, then V' is a vertex cover for G. Further, $|V'| \leqslant k$ since J contains a total of $|V| + 3|E| + k$ vertices, and in J there are $3|E|$ vertices per edge $e \in E$ (for $R_e, R_{e,u}, R_{e,v}$), and at least one vertex for each $v \in V$. □

In Fig. 13, we present a greedy heuristic for computing a well formed join graph with small error. Algorithm `CoalesceJoinGraphs`, in each iteration of the outermost while loop, merges the pair of vertices in J that causes the error to be minimum, until the error cannot be reduced any further by coalescing vertices. Algorithm `ComputeSpace`, is used to compute the average (Section 4.1) or maximum error (Section 4.2) for a join graph. Also, in order to ensure that graph J always stays well formed, J is initially set to be equal to the set of all the individual join graphs for queries in $\mathcal{Q}$. In each subsequent iteration, only

Procedure: CoalesceJoinGraphs($\mathcal{Q}, M$)
Require: $\mathcal{Q}$ is query workload, M is available memory.
Ensure: Returns a well-formed join graph $\mathcal{J}(\mathcal{Q})$.
1: $J = \cup_{Q \in \mathcal{Q}} \mathcal{J}(Q)$
2: (m, err) = ComputeSpace(J, M)
3: *flag* = *true*
4: **while** (*flag* = *true*) **do**
5: *cur_err* = ∞
6: *flag* = *false*
7: **for all** pairs of vertices v_i, v_j in J such that $R(v_i) = R(v_j)$ **and** $\mathcal{A}(v_i) = \mathcal{A}(v_j)$ **do**
8: Let J' be the join graph after v_i and v_j are coalesced in J
9: (m', err') = ComputeSpace(J', M)
10: **if** ($err' < cur\_err$ **and** J' is well-formed) **then**
11: *cur_err* = err'
12: *cur_J* = J'
13: **end if**
14: **end for**
15: **if** ($cur\_err \leq err$) **then**
16: *err* = *cur_err*
17: J = *cur_J*
18: *flag* = *true*
19: **end if**
20: **end while**
21: return (J, ComputeSpace(J, M))

Fig. 13. Algorithm `CoalesceJoinGraphs`.

vertices for identical relations that have the same attribute sets and preserve the well-formedness of J are coalesced. Note that checking whether graph J' is well formed in Step 10 of the algorithm can be carried out very efficiently, in time proportional to the number of edges in J'. Well-formedness testing essentially involves partitioning the edges of J' into equivalence classes, each class consisting of ξ-equivalent edges, and then verifying that no equivalence class contains multiple edges from the same join query. Also, observe that `CoalesceJoinGraphs` makes at most $O(N^3)$ calls to `ComputeSpace`, where N is the total number of vertices in all the join graphs $\mathcal{J}(Q)$ for the queries, and this determines its time complexity as $O(N^3 \min\{|\mathcal{Q}|, |J|\} \cdot (|\mathcal{Q}| + |J|)^3)$.

6. Experimental study

In this section, we present the results of an experimental study of our sketch-sharing algorithms for processing multiple `COUNT` queries in a streaming environment. Our experiments consider a wide range of `COUNT` queries based on the TPC-H benchmark, and with synthetically generated data sets. The reason we use synthetic data sets is that these enable us to measure the effectiveness of our sketch sharing techniques for a variety of different data

distributions and parameter settings. The main findings of our study can be summarized as follows.

- *Effectiveness of sketch sharing.* Our experiments with the TPC-H query workload indicate that, in practice, sharing sketches among queries can significantly reduce the number of sketches needed to compute estimates. This, in turn, results in better utilization of the available memory, and much higher accuracy for returned query answers. For instance, for the TPC-H query set, the number of vertices in the final coalesced join graph returned by our sketch-sharing algorithms decreases from 34 (with no sharing) to 16. Further, even with $W_Q = 1$ (for all queries Q), compared to naive solutions which involve no sketch sharing, our sketch-sharing solutions deliver improvements in accuracy ranging from a factor of 2–4 for a wide range of multi-query workloads.
- *Benefits of intelligent space allocation.* The errors in the approximate query answers computed by our sketch-sharing algorithms are smaller if approximate weight information $W_Q = 8\mathrm{Var}[X]/E[X]^2$ for queries is available. Even with weight estimates based on coarse statistics on the underlying data distribution (e.g., histograms), accuracy improvements of up to a factor of 2 can be obtained compared with using uniform weights for all queries.

Thus, our experimental results validate the thesis of this paper that sketch sharing can significantly improve the accuracy of aggregate queries over data streams, and that a careful allocation of available space to sketches is important in practice.

6.1. Experimental testbed and methodology

Algorithms for answering multiple aggregate queries. We compare the error performance of the following two sketching methods for evaluating query answers.

- *No sketch sharing.* This is the naive sketching technique from Section 2.2 in which we maintain separate sketches for each individual query join graph $\mathcal{J}(Q)$. Thus, there is no sharing of sketching space between the queries in the workload, and independent atomic sketches are constructed for each relation, query pair such that the relation appears in the query.
- *Sketch sharing.* In this case, atomic sketches for relations are reused as much as possible across queries in the workload for the purpose of computing approximate answers. Algorithms described in Sections 4 and 5 are used to compute the well formed join graph for the query set and sketching space allocation to vertices of the join graph (and queries) such that either the average-error or maximum-error metric is optimized. There are two solutions that we explore in our study, based on whether prior (approximate) information on join and self-join sizes is available to our algorithms to make more informed decisions on memory allocation for sketches.
  - No prior information. The weights for all join queries in the workload are set to 1, and this is the input to our sketch-sharing algorithms.
  - Prior information is available. The ratio $8\mathrm{Var}(X)/E[X]^2$ is estimated for each workload query, and is used as the query weight when determining the memory to be allocated to each query. We use coarse one-dimensional histograms for each relational attribute to estimate join and self-join sizes required for weight computation. Each histogram is given 200 buckets, and the frequency distribution for multi-attribute relations is approximated from the individual attribute histograms by applying the attribute value independence assumption.

Query workload. The query workloads used to evaluate the effectiveness of sketch sharing consist of collections of JOIN-COUNT queries from the TPC-H benchmark (Table 2). Fig. 14 depicts a subset of the tables in the TPC-H schema, and the edges represent the attribute equi-join relationships between the tables. We did not consider the tables NATION and REGION since the domain sizes for both are very small (25 and 5, respectively). We consider three query workloads, each consisting of a subset of queries shown in Fig. 2. In the figure, each query is described in terms of the equi-join constraints it contains; further, except for equi-join constraints, we omit all other selection conditions/constraints from the query WHERE clause. The first workload consists of queries Q_1 through Q_{12}, which are the standard TPC-H benchmark join queries (restricted to only contain equi-join constraints). In order to get a feel for the benefits of sketch sharing as the degree of sharing is increased, we consider a second query workload containing all the queries Q_1 to Q_{29}.

Table 2
Workload queries

Q_1	1, 2	Q_9	1	Q_{17}	8, 9	Q_{25}	2, 7
Q_2	4, 5	Q_{10}	6, 7	Q_{18}	5, 9	Q_{26}	1, 6
Q_3	3, 4, 5	Q_{11}	5, 8	Q_{19}	6, 8	Q_{27}	3, 8
Q_4	4, 5, 8	Q_{12}	10	Q_{20}	7, 8	Q_{28}	1, 2, 3
Q_5	4, 5, 8, 9	Q_{13}	4	Q_{21}	8	Q_{29}	2, 3, 4
Q_6	2	Q_{14}	3	Q_{22}	6		
Q_7	5	Q_{15}	3, 4	Q_{23}	7		
Q_8	9	Q_{16}	5, 8	Q_{24}	2, 3		

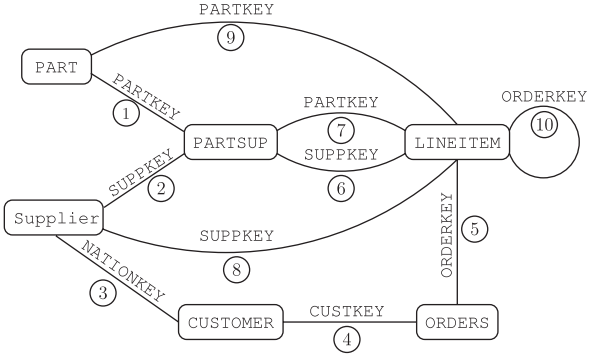

Fig. 14. Relations, join attributes and equi-join constraints for TPC-H schema.

Observe that workload 2 contains a larger number of queries over the same relations, and so we expect to see better improvements from sketch sharing for workload 2 compared to workload 1. Finally, workload 3 contains queries Q_6 to Q_{12} and Q_{28}. We use this workload to demonstrate the accuracy gains obtained as a result of using nonuniform query weights. In our experiments, we did not realize much benefit from taking into account approximate query weights for workloads 1 and 2. This is because both workloads contain queries with large weights that are distributed across all the relations. These heavy queries determine the amount of sketching space allotted to the underlying relations, and the results become very similar to those for uniform query weights.

Data set. We used the synthetic data generator from [26] to generate the relations shown in Fig. 14. The data generator works by populating uniformly distributed rectangular regions in the multi-dimensional attribute space of each relation. Tuples are distributed across regions and within regions using a Zipfian distribution with values z_{inter} and z_{intra}, respectively. We set the parameters of the data generator to the following default values: size of each domain = 1024, number of regions = 10, volume of each region = 1000–2000, skew across regions (z_{inter}) = 1.0, skew within each region (z_{intra}) = 0.0–0.5 and number of tuples in each relation = 10,000,000.

Answer-quality metrics. In our experiments we use the square of the absolute relative error $((actual\text{-}approx)^2/actual^2)$ in the aggregate value as a measure of the accuracy of the approximate answer for a single query. For a given query workload, we consider both the average-error and maximum-error metrics, which correspond to averaging over all the query errors and taking the maximum from among the query errors, respectively. We repeat each experiment 100 times, and use the average value for the errors across the iterations as the final error in our plots.

6.2. Experimental results

Results: *sketch sharing.* Figs. 15–18 depict the average and maximum errors for query workloads 1 and 2 as the sketching space is increased from 2K to 20K words. From the graphs, it is clear that with sketch sharing, the accuracy of query estimates improves. For instance, with workload 1, errors are generally a factor of two smaller with sketch sharing. The improvements due to sketch sharing are even greater for workload 2 where due to the larger number of queries, the degree of sharing is higher. The improvements can be attributed to our sketch-sharing algorithms which drive down the number of join graph vertices from 34 (with no sharing) to 16 for workload 1, and from 82 to 25 for workload 2. Consequently, more sketching space can be allocated to each vertex, and hence the accuracy is better with sketch sharing compared to no sharing. Further, observe that in most cases, errors are less than 10% for sketch sharing, and as would be expected, the accuracy of estimates gets better as more space is made available to store sketches.

Results: *intelligent space allocation.* We plot in Figs. 19 and 20, the average and maximum error graphs for two versions of our sketch-sharing algorithms, one that is supplied uniform query weights, and another with estimated weights computed using coarse histogram statistics. We considered query workload 3 for this experiment since workloads 2 and 3 have queries with large weights that access all the underlying relations. These queries tend to dominate in the space allocation procedures, causing the final result to be very similar to the uniform query weights case. But with workload 3, query Q_{29} has a considerably larger weight than other queries in the workload (since it has three equi-joins), and so our space allocation algorithms are more effective and allocate more space to Q_{29}. Thus, with intelligent space allocation, even with coarse statistics on the data

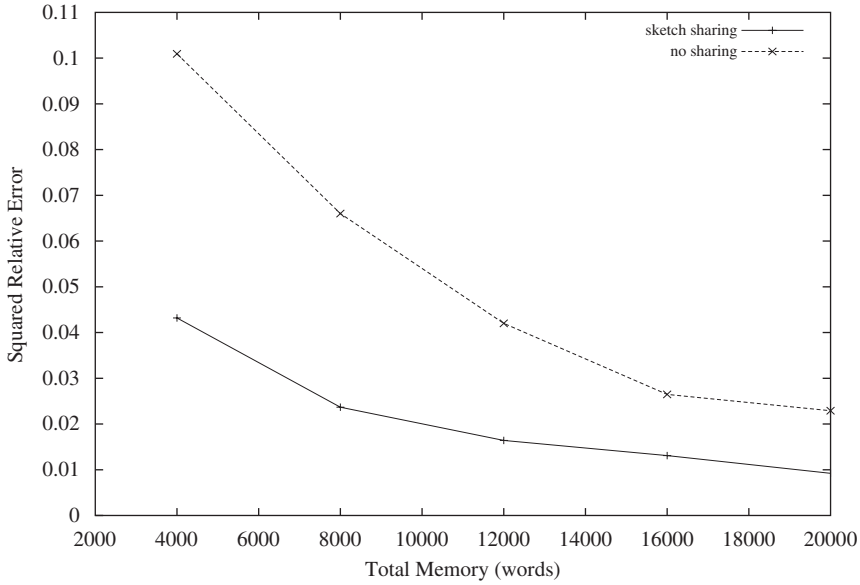

Fig. 15. Average error (workload 1).

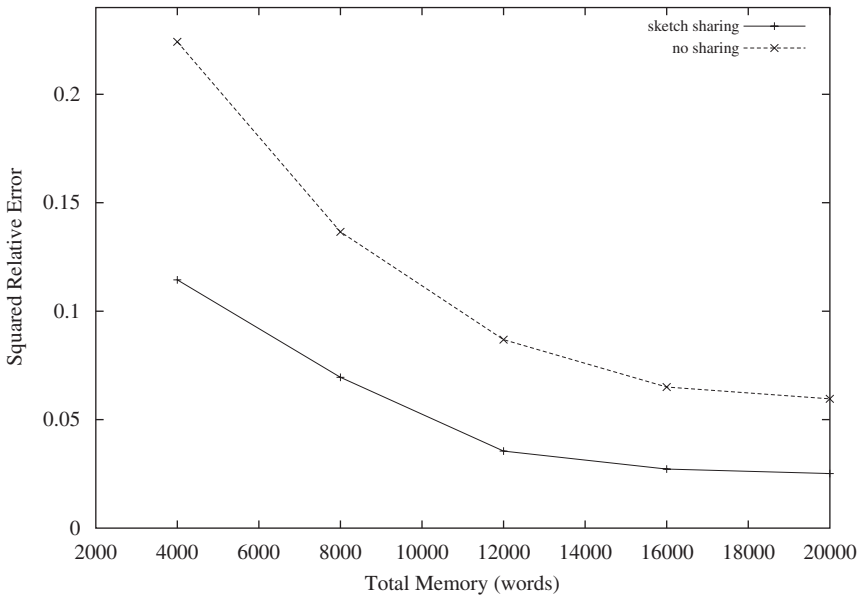

Fig. 16. Maximum error (workload 1).

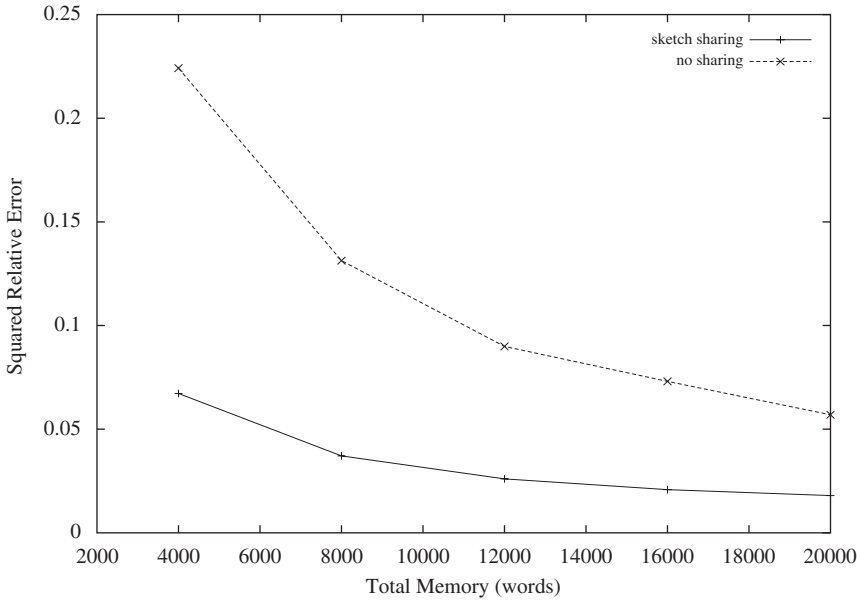

Fig. 17. Average error (workload 2).

distribution, we are able to get accuracy improvements of up to a factor of 2 by using query weight information.

Accommodating queries with selection predicates. In the above experiments, the selection predicates have been ignored since they are not explicitly supported by sketches. In the case when a single query is evaluated, the selection can be performed before the sketching of the tuples, thus effectively incorporating the selection. The same technique is possible when multiple queries are present, but by doing so the opportunity for sharing is reduced since the sketch of a relation cannot be straightforwardly shared if different selection predicated are used. Sharing is still possible though for the relations without selection predicates. In the case of TPCH, only the sketches for `PARTSUP` and `LINEITEM` can be shared. While the sharing opportunities are reduced in this case for TPCH, in other applications in which a large number of queries are present but with few selection predicates (processing networking data comes to mind), significant sharing opportunities might still be present. An interesting but nontrivial question is whether the sketches can be shared when two different selection predicates are required. We plan to address this issue in future work.

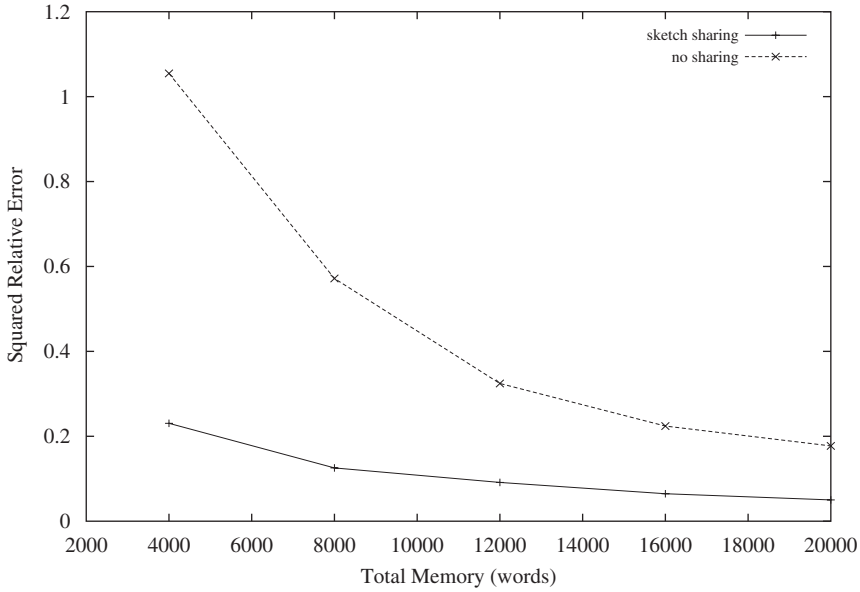

Fig. 18. Maximum error (workload 2).

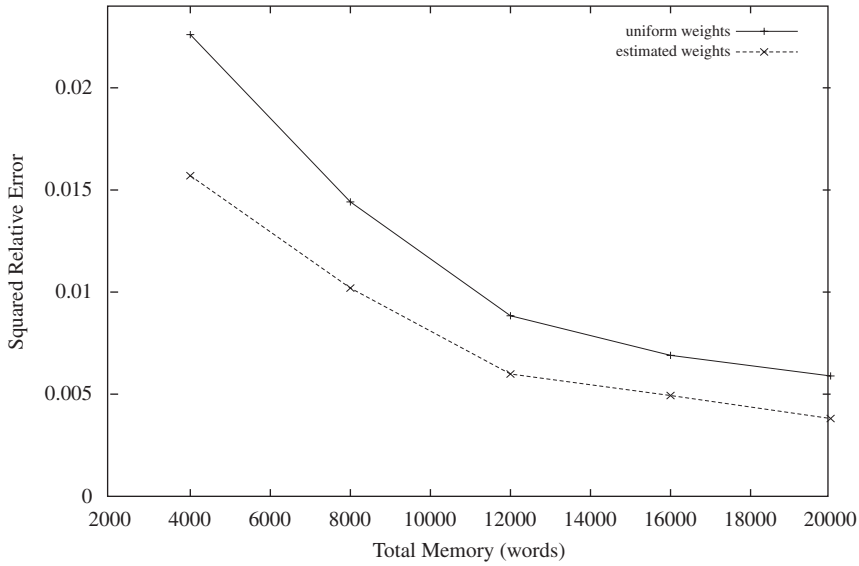

Fig. 19. Average error (workload 3).

7. Concluding remarks

In this paper, we investigated the problems that arise when data-stream sketches are used to process *multiple* aggregate SQL queries concurrently. We provided necessary and sufficient conditions for multi-query sketch sharing that guarantee the correctness of the result-estimation process, and we developed solutions to the optimization problem of determining sketch-sharing configurations that are optimal under average and maximum error metrics for a given amount of space. We proved that the problem of optimally allocating space to sketches such that query estimation errors are minimized is $\mathcal{NP}$-hard. As a result, for a given multi-query workload, we developed a mix of near-optimal solutions (for space allocation) and heuristics to compute the final set of sketches that result in small errors. We conducted an experimental study with query workloads from the TPC-H benchmark; our findings indicate that (1) compared to a naive solution that does not share sketches among queries, our sketch-sharing solutions deliver improvements in accuracy ranging from a factor of 2–4, and (2) The use of prior information about queries (e.g., obtained from coarse histograms) increases the effectiveness of our

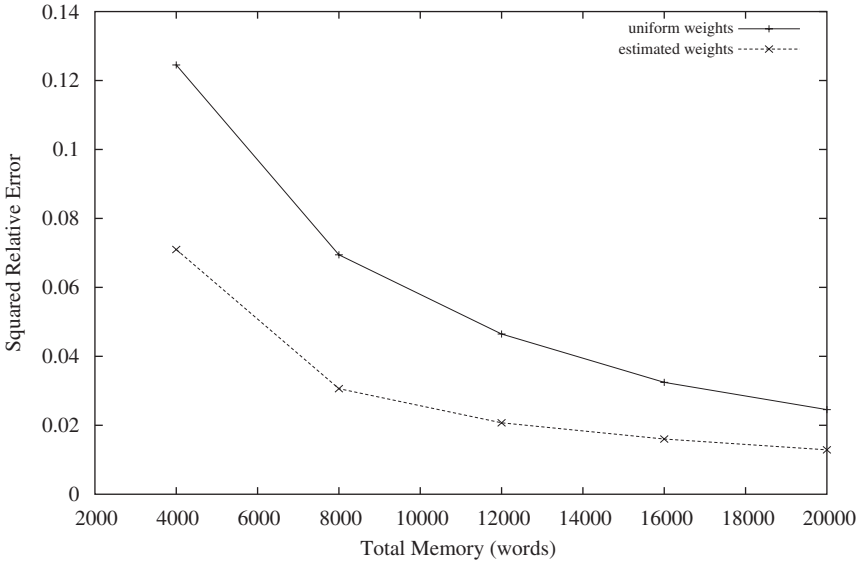

Fig. 20. Maximum error (workload 3).

memory allocation algorithms, and can cause errors to decrease by factors of up to 2.

References

[1] M. Greenwald, S. Khanna, Space-efficient online computation of quantile summaries, in: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, May 2001.

[2] A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M.J. Strauss, How to summarize the universe: dynamic maintenance of quantiles, in: Proceedings of the 28th International Conference on Very Large Data Bases, Hong Kong, China, August 2002.

[3] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, L. Trevisan, Counting distinct elements in a data stream, in: Proceedings of the 6th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM'02), Cambridge, Massachusetts, September 2002.

[4] P.B. Gibbons, S. Tirthapura, Estimating simple functions on the union of data streams, in: Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, Crete Island, Greece, July 2001.

[5] P.B. Gibbons, Distinct sampling for highly-accurate answers to distinct values queries and event reports, in: Proceedings of the 27th International Conference on Very Large Data Bases, Rome, Italy, September 2001.

[6] M. Charikar, K. Chen, F.-C. Martin, Finding frequent items in data streams, in: Proceedings of the International Colloquium on Automata, Languages, Programming, Malaga, Spain, July 2002.

[7] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proceedings of the 28th International Conference on Very Large Data Bases, Hong Kong, China, August 2002.

[8] N. Alon, P.B. Gibbons, Y. Matias, M. Szegedy, Tracking join and self-join sizes in limited storage, in: Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Philadelphia, Pennsylvania, May 1999.

[9] N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in: Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, May 1996, pp. 20–29.

[10] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan, An approximate L^1-difference algorithm for massive data streams, in: Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, New York, NY, October 1999.

[11] P. Indyk, Stable distributions, pseudorandom generators, embeddings and data stream computation, in: Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science, Redondo Beach, California, November 2000, pp. 189–197.

[12] P. Domingos, G. Hulten, Mining high-speed data streams, in: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Boston, Massachusetts, August 2000, pp. 71–80.

[13] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan, Clustering data streams, in: Proceedings of the 2000 Annual Symposium on Foundations of Computer Science (FOCS), November 2000.

[14] J. Gehrke, F. Korn, D. Srivastava, On computing correlated aggregates over continual data streams, in: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, September 2001.

[15] A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M.J. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in: Proceedings of the 27th International Conference on Very Large Data Bases, Rome, Italy, September 2000.

[16] N. Thaper, S. Guha, P. Indyk, N. Koudas, Dynamic multidimensional histograms, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 2002.

[17] A. Arasu, B. Babcock, S. Babu, J. McAlister, J. Widom, Characterizing memory requirements for queries over continuous data streams, in: Proceedings of the 21st ACM Symposium on Principles of Database Systems, Madison, Wisconsin, June 2002.

[18] M. Garofalakis, J. Gehrke, R. Rastogi, Querying and mining data streams: you only get one look, Tutorial in 28th International Conference on Very Large Data Bases, Hong Kong, China, August 2002.

[19] A. Dobra, M. Garofalakis, J. Gehrke, R. Rastogi, Processing complex aggregate queries over data streams, in: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, Washington, June 1998, pp. 448—459.

[20] T.K. Sellis, Multiple-query optimization, ACM Trans. Database Syst. 13 (1) (1988) 23–52.

[21] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe, Efficient and extensible algorithms for multi query optimization, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, May 2000.

[22] J. Chen, D. DeWitt, F. Tian, Y. Wang, NiagaraCQ: a scalable continuous query system for internet databases, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, May 2000.

[23] S.M. Stefanov, Separable programming, in: Applied Optimization, vol. 53, Kluwer Academic Publishers, Dordrecht, 2001.
[24] T. Ibaraki, N. Katoh, Resource Allocation Problems—Algorithmic Approaches, MIT Press Series in the Foundations of Computing, 1988.
[25] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, Massachusetts, 1990.
[26] J.S. Vitter, M. Wang, Approximate computation of multidimensional aggregates of sparse data using wavelets, in: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, May 1999.